Software (In)Security of Smart Contracts and Trusted Enclaves

Dissertation
zur Erlangung des Doktorgrades

„Dr. rer. nat.“

der Fakultät für Wirtschaftswissenschaften
der Universität Duisburg-Essen

vorgelegt von

Michael Rodler

aus
Salzburg, Österreich

Betreuer:
Prof. Dr.-Ing. Lucas Vincenzo Davi
Lehrstuhl für Systemsicherheit

Essen, März 2023

DOI: 10.17185/duepublico/81316
URN: urn:nbn:de:hbz:465-20240109-121727-7

1. Gutachter:
Univ.-Prof. Dr.-Ing. Lucas Vincenzo Davi
Universität Duisburg-Essen

2. Gutachter:
Univ.-Prof. Dr. Ghassan Karame
Ruhr-Universität Bochum

Tag der mündlichen Prüfung: 27. Juni 2023

ABSTRACT

Secure execution environments promise developers a way to secure their software even when running in untrusted environments. For example, the Intel software guard extensions (SGX) technology strives to provide confidential computing on hardware operated by an untrusted party. In Ethereum, software called smart contracts manages a large number of funds and assets. As such, correct execution of smart contracts is paramount. However, arbitrary network participants execute the smart contracts within an open distributed system. Secure execution environments ensure with cryptographic protocols that the integrity and confidentiality of both execution and data are preserved. While these new execution environments offer many security guarantees, they cannot automatically secure the software executing within the execution environment. This dissertation challenges the security of the software developed for and running within secure execution environments. More specifically, we tackle the (in)security of software for the execution environments of the SGX trusted computing technology and the Ethereum blockchain system. This dissertation develops methods for identification and remediation of security vulnerabilities specific to the respective secure execution environment.

Existing methods for automated vulnerability identification are not sufficient as they are not tailored to the respective execution environments. In this dissertation, we identify these shortcomings and develop new automated analysis methods. Here, an analysis tool is developed that adapts generic symbolic execution to identify vulnerabilities at the boundary between SGX enclave code and the untrusted, and as such potentially attacker-controlled, host system. Furthermore, this work shows how to use the fuzzing method to efficiently identify even complex vulnerability patterns, such as reentrancy, in Ethereum smart contracts.

Identifying vulnerabilities alone is not sufficient to ensure the security of already deployed software. As such, this dissertation also covers new methods for vulnerability remediation. Due to the immutability of smart contracts in Ethereum, it is especially hard to fix bugs and vulnerabilities. As such, we discuss a taint-tracking-based detection of reentrancy attacks, which can be integrated directly into the blockchain system. Furthermore, we also discuss an approach to automate large parts of the patching process of a smart contract, allowing developers to securely operate a smart contract on Ethereum.

ZUSAMMENFASSUNG

Sichere Ausführungsumgebungen bieten Entwicklern die Möglichkeit, ihre Software abzusichern, auch wenn sie in nicht vertrauenswürdigen Umgebungen ausgeführt wird. Die SGX Technologie von Intel versucht, vertrauliche Datenverarbeitung (*confidential computing*) auf Hardware zu ermöglichen, die von einer nicht vertrauenswürdigen Partei betrieben wird. In Ethereum verwaltet Software, so genannte *Smart Contracts*, Cryptocurrency und anderen Vermögenswerten. Deswegen ist die korrekte Ausführung von Smart Contracts von größter Bedeutung. Da jeder dem Ethereum Netzwerk beitreten kann, werden Smart Contracts durch beliebige Netzwerkteilnehmer ausgeführt. Mithilfe von kryptografischen Protokollen sorgen sichere Ausführungsumgebungen dafür, dass die Integrität und Vertraulichkeit der Ausführung und der Daten gewahrt wird. Diese neuen Ausführungsumgebungen bieten zwar viele Sicherheitsgarantien, aber sie können sie die darin ausgeführte Software nicht vollständig vor Schwachstellen schützen. Diese Dissertation untersucht die Sicherheit der Software, die für sichere Ausführungsumgebungen entwickelt wurde, genauer gesagt mit der (Un-)Sicherheit von Software für die SGX Trusted Computing Technologie und dem Ethereum Blockchain System. Dabei werden speziell angepasste Methoden zur Identifizierung und Behebung von Sicherheitslücken erforscht.

Bestehende Methoden zum automatisierten Auffinden von Schwachstellen sind nicht ausreichend, da sie nicht auf die jeweiligen Ausführungsumgebungen zugeschnitten sind. In dieser Dissertation identifizieren wir diese Defizite und entwickeln neue Analysemethoden. Dabei wird ein Analysewerkzeug vorgestellt, welches generische symbolische Ausführung adaptiert, um Schwachstellen an der Grenze zwischen SGX Enclave-Code und dem nicht vertrauenswürdigen Host-System aufzudecken. Darüber hinaus zeigt diese Arbeit, wies sogenanntes Fuzzing eingesetzt werden kann, um selbst komplexe Schwachstellen automatisiert zu finden, wie z.B. Reentrancy in Ethereum Smart Contracts. Da das Auffinden von Schwachstellen allein jedoch nicht ausreicht, um die Sicherheit von bestehender Software zu gewährleisten, werden in dieser Dissertation auch neue Methoden zur Behebung von Schwachstellen untersucht. Aufgrund der Unveränderlichkeit von Smart Contracts in Ethereum ist es besonders schwierig Softwarefehler zu beheben. Wir diskutieren daher eine auf Taint-Tracking basierende Erkennung von Reentrancy-Angriffen, die in das Blockchain-System integriert wird und Schwachstellen zur Laufzeit erkennt und blockiert. Außerdem diskutieren wir eine Methode, um große Teile des Patching-Prozesses von Smart Contracts zu automatisieren, wodurch es Entwicklern ermöglicht wird, einen Smart Contract auf Ethereum sicher zu betreiben.

ACKNOWLEDGEMENTS

I would like to thank my advisor Lucas Davi for giving me the opportunity to pursue a PhD, for teaching me how to put all my thoughts about software security together into coherent and readable sentences, and for directing me to work on those problems that turned out to be quite fruitful research directions. I would also like to thank my colleagues, Sebastian, Tobias, Jens-Rene, David, Oussama, and Christian for the inspiring and fun discussions during coffee and lunch breaks. Many thanks also go out to the various people, with whom I collaborated over the years: Thorsten Holz, Ghassan Karame, Wenting Li, and Lukas Bernhard. It was an honor and a pleasure working with you!

On the more personal side, I am deeply grateful to my parents, Waltraud and Hannes, who nurtured my curiosity and always supported me no matter what. Thank you for this: without your support I would not be where I am today. Above all, I would like to thank my partner Marion with all my heart. You supported me on my academic journey, forced me to a somewhat reasonable work-life balance, and also endured the occasional crunch-time right before deadlines. For all this and much more: Thank you! <3

CONTENTS

CHAPTER 1

INTRODUCTION

Software security is paramount to today's infrastructure, encompassing more than traditional Information Technology (IT) domains. For example, the cyber-physical system (CPS) domain encompasses a blend of the IT and Operational Technology (OT) domains, where software components directly control physical processes. Since the infamous *Stuxnet* incident, it has become clear that software is a primary attack vector against highly critical systems. Similarly, the financial industry has become increasingly automated, with extremes such as automated high-frequency trading or the rise of the decentralized finance (DeFI) industry.

However, software security still poses a major problem for software vendors. Design flaws and programming mistakes are common during software development, making pieces of software vulnerable to attacks. For example, today, 34 years after the first large-scale exploitation of a so-called *buffer overflow* issue [See] and 26 years after the first public article by One [One96], large C/C++ software projects are still being actively exploited using *memory corruption exploits* [Zer]. In spite of research on defenses and mitigations both in industry and academia [Sze+13], memory corruption is still a major issue. For example, Microsoft, arguably one of the largest software vendors, still identifies memory corruption as the major source of security vulnerabilities in their products [Tho19]. However, software security is not only a problem for the systems software domain. A new set of vulnerability types accommodates every new programming paradigm. For example, various forms of injection attacks plague software running web applications. Most recently, smart contracts, software executed as part of blockchain protocols, are targeted by attackers using software vulnerabilities.

Modern IT systems are designed to minimize the *trusted computing base (TCB)*, thereby reducing the potential impact of programming errors in critical components. Anderson defines the TCB as follows [And20, p 320]:

> More formally, the TCB is defined as the set of components (hardware, software, human, ...) whose correct functioning is sufficient to ensure that the security policy is enforced, or, more vividly, whose failure could cause a breach of the security policy.

Minimizing the TCB follows the intuition that smaller systems have fewer chances to contain bugs that allow an attacker to break the system's security policy, i.e., they have smaller attack surfaces. Furthermore, smaller systems are more amenable to formal analysis, code review, and testing, thereby increasing the probability of spotting security vulnerabilities upfront. As such, almost all current information systems attempt to reduce their TCB.

However, several recent technologies attempt to make it possible to further minimize the TCB. Traditionally, the TCB would consist of the hardware, the operating system kernel, and several trusted system services. However, the *trusted execution environments (TEEs)* remove many of these traditional components from the TCB. The operating system is no longer part of the TCB. Most TEEs remove even the human operators of a device from the TCB. The only remaining trusted party is the hardware vendor. The TCB is reduced to the central processing unit (CPU) and its directly connected components. For example, in SGX, anything outside the physically sealed CPU die, including main memory, is considered untrusted [CD16]. This scenario is particularly appealing for the cloud setting, where a user wants to offload computation to a remote system operated by a potentially untrusted party.

A second example of a highly reduced TCB is the smart contract as used in blockchain ecosystems [Woo19]. A decentralized application (DApp) is backed by a smart contract, which encodes the *business logic* of the application: essentially reducing the TCB to the smart contract and the blockchain protocol. Any interaction with the smart contract is cryptographically secured, and its execution is replicated among the entire distributed blockchain system. The correct execution of the smart contract is ensured using economic incentives inherent to the distributed system. For example, an attacker would have to subvert at least 51 % of the computing power of a classical *Proof-of-Work* blockchain system to manipulate ledger entries, i.e., results of a smart contract's execution [Nak08]. Furthermore, due to the decentralized nature and diversity of participating nodes in the blockchain network, the software running and executing the blockchain protocols is not necessarily part of the TCB. However, due to the replicated nature—and in contrast to TEEs—this comes at the cost of confidentiality. For example, in Ethereum, all smart contracts and their associated data are public. As a consequence, one has to resort to more complex cryptographic protocols to ensure data confidentiality.

This dissertation examines software systems, which are summarized under the umbrella term *secure execution environments* in the context of this dissertation. We define secure execution environments as execution environments that exhibit two main properties: (1) Secure execution environments exhibit a significantly reduced TCB compared to classical software systems. (2) Code and data integrity is protected by the execution environment. However, to be useful, such secure execution environments must allow untrusted—and potentially malicious—parties to interact with the software inside the secure execution environment. While the secure execution environment protects the software running inside TEEs from manipulation, there is still a remaining attack surface accessible through the interface between trusted and untrusted software. Software running inside a secure execution environment must exercise *self-protection* against malicious inputs. This leads to the central research question of this dissertation:

How can we secure the software running inside secure execution environments?

To answer this central research question, this dissertation investigates two orthogonal directions: identifying vulnerabilities and hardening software. This dissertation describes novel methods to automatically identify vulnerabilities in software targeting secure execution environments. Developers can utilize the methods and tools developed as part of this dissertation to identify vulnerabilities before deployment. Similarly, users can assess the security of deployed software before they decide to trust it with their data. Applying these vulnerability detection techniques to current systems, we demonstrate that software, which relies on secure execution environments, is not as secure as previously assumed. Our analysis of the discovered issues shows that many security vulnerabilities are due to programming mistakes that can be traced back to misunderstandings of the constraints imposed by the underlying secure execution environment. To improve the security of software, this dissertation shows approaches to securing software within secure execution environments. We tackle both developing a robust process for patching software in secure execution environments and hardening the secure execution environment itself.

To study the security of software inside secure execution environments, we analyze two highly relevant technologies implementing such an environment. Ethereum is the first, most prominent, and most popular Blockchain platform natively supporting smart contracts. As such, Ethereum has become the backbone of the upcoming DeFI industry. In Ethereum and other blockchain systems, smart contracts can own and autonomously transfer currency to other parties. As such, smart contracts must execute correctly and satisfy the intention of all stakeholders. While the Ethereum system guarantees that the smart contract code is executed correctly with the consensus protocol, it does not protect smart contracts against attacks on a logical level, i.e., exploitable software bugs. Over the last decade, the blockchain community has witnessed several major bugs and vulnerabilities within smart contracts. In many cases, these vulnerabilities allowed attackers to gain incredible amounts of cryptocurrency. In 2016 the infamous attack on the "The DAO" smart contract resulted in a loss of Ether, worth over 50 million US Dollars at the time the attack occurred [Pri16]. Still, in 2020 the community has witnessed attacks against the *Uniswap V2* and *Lendf.me* contracts [Tor+21b], where the attackers gained Tokens worth roughly 25 million US Dollars. These attacks were due to the same type of software bug: an exploitable reentrancy issue. This continuous stream of incidents due to smart contract bugs demonstrates that practical software security for smart contracts is still a highly relevant and open research area.

Intel introduced SGX as its most sophisticated incarnation of trusted computing technology [Hoe+13; Intel4; McK+13] to date. SGX shields so-called *enclaves*, the trusted code in the SGX model, from malicious access by the operator of the machine, other applications, the underlying operating system, and the hypervisor. The only trusted component in the SGX setting is the Intel CPU package. Additionally, SGX features many well-known concepts from earlier trusted computing technology, such as data binding, sealing, and remote attestation. Combined, all these features make SGX a prime candidate for ensuring data confidentiality and integrity in the public cloud [BPH14; Sch+15a]. The SGX technology ensures that the user can establish a

secure channel into the SGX enclave while simultaneously ensuring that the enclave is in a trustworthy state. Using the secure channel, the user can safely transfer data into the enclave executing on public cloud hardware. Similarly, SGX can be used to protect, e.g., biometric data processing, on mobile endpoints such as laptops. For example, the SGX technology is used in fingerprint sensor software (Section 4.3), DRM protection [Cyb], and privacy-preserving applications like Signal [Mar17]. As such, SGX is currently one of the most prominent trusted computing technologies.

Current development practices for software in—the still relatively new—secure execution environments are significantly lacking. Software developers are not as familiar with the constraints of the programming models of secure execution environments, leading to fatal programming mistakes that can be abused by attackers [ABC17; Luu+16; Van+19]. It is important to detect such attacks early in the development lifecycle of software written for secure execution environments. We investigate offensive methods to identify security issues automatically and before deployment. To this end, we identify symbolic execution and fuzzing as two industry-standard methods for automatically uncovering vulnerabilities in software. However, these methods need to be adapted to the particularities of (secure) execution environments to become effective tools in the hands of developers. Equally important, we also investigate defensive methods as a second direction in this dissertation. We investigate the capability of the secure execution environment to detect and block attacks as they are happening as a second line of defense. Such built-in defensive methods allow the software to remain secure even in the presence of a certain type of programming mistake that would otherwise lead to a vulnerability. We also investigate patching software in secure execution environments, which is highly challenging since the software vendor has only limited control over the deployment of the software. For example, a software vendor of a classical web service can take down the service until they finish developing a patch. However, a software vendor for a secure execution environment usually cannot do this, as the software is either already deployed to an untrusted client system (TEE) or is replicated among a decentralized system (Blockchain).

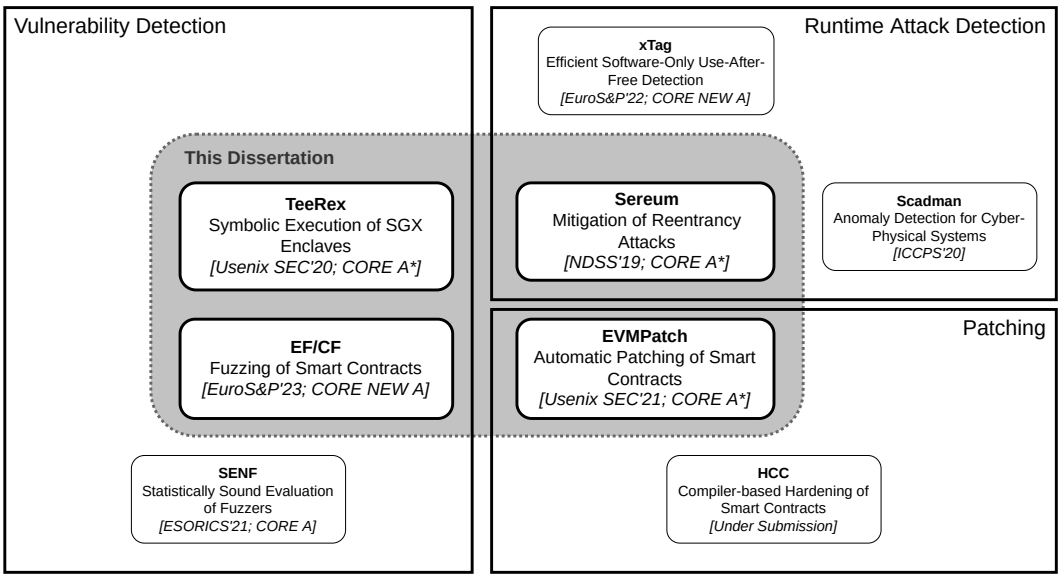


Figure 1.1: Overview on the four main publications of this dissertation and additional publications grouped according to research direction.

Outline and Contributions

This dissertation is structured into three major parts: a discussion of the background and literature (Chapters 2 and 3), followed by a discussion of two orthogonal aspects of software security for secure execution environments: (1) identifying (Chapters 4 and 5) and (2) mitigating vulnerabilities (Chapters 6 and 7).

First, we give the necessary technical background on software vulnerabilities in general and on techniques to automatically identify vulnerabilities using dynamic analysis methods, such as symbolic execution and fuzzing, in Chapter 2. Then, we discuss the technical details of the two secure execution environments covered by this dissertation in Chapter 3: Consensus-enforced distributed computation with *Ethereum* and hardware-secured confidential and trusted computation with *SGX*. Both architectures and programming models allow untrusted users to interact with a trusted software component using an interface similar to normal library code components.

Figure 1.1 shows an overview of the research topics and contributions to the field of software security by the author. Concerning vulnerability detection, we present two new approaches to automatically identifying security vulnerabilities using symbolic execution of SGX enclaves (Chapter 4) and fuzzing of smart contracts (Chapter 5). In the *SENF* project [Paa+21a], we develop a statistically sound way of comparing fuzzers, which we utilize to evaluate our newly developed smart contract fuzzer EF$\flat$CF in Chapter 5. In Chapter 6, we present the first taint-tracking-based dynamic analysis for smart contracts, dubbed Sereum, to detect and mitigate reentrancy attacks at runtime. The *xTag* project presents an efficient detection approach for *use-after-free* exploit attempts on legacy *x86* systems [Ber+22]. With *Scadman*, we develop an approach to anomaly detection in CPSs using differential execution with a simulated CPS. With EVMPatch in Chapter 7, we show how to streamline and automate the

patching process of smart contracts, an otherwise cumbersome and error-prone process. *HCC* is a complementary patching approach that uses source-to-source compilation to introduce hardening checks [Gie+22]. In the following, we give an outline of the main chapters of this dissertation, their contributions, and prior publications of the author.

Finding Vulnerabilities in SGX Enclaves using Symbolic Execution In Chapter 4, we describe an approach to identify vulnerabilities in SGX enclaves using symbolic execution. In contrast to prior studies on SGX enclaves, we present a methodology that allows us to systematically search for security vulnerabilities in real-world enclave code. First, we show how to adapt symbolic execution to SGX enclave code, tackling several challenges imposed by the SGX technology. Second, we focus on identifying issues in the interface between the untrusted software components and the trusted SGX enclave code. To achieve this, we show how to model this interface between trusted and untrusted software in a symbolic execution engine. We perform a study to assess the state of public SGX enclaves. Using our symbolic execution engine, we identify several programming mistakes that are due to misconceptions with respect to constraints imposed by SGX. We present a root-cause analysis and systematization of the discovered bugs and deduce several secure programming guidelines for SGX enclaves from our findings.

This chapter is based on the following publication, which has been nominated as a finalist in the CSAW Applied Research competition and was awarded the 3rd place in the 8. German IT-Security Price 2020:
"TeeRex: Discovery and Exploitation of Memory Corruption Vulnerabilities in SGX Enclaves". ***29th USENIX Security Symposium*, 2020**. Tobias Cloosters, Michael Rodler, and Lucas Davi

Adapting and Improving Fuzzing for Smart Contracts Chapter 5 shows how to adapt state-of-the-art fuzzing techniques to smart contracts and significantly improve the field of automated smart contract analysis. In contrast to other prior fuzzing approaches, the approach presented in this chapter emphasizes high test case throughput. This allows our fuzzer to quickly search for potentially interesting inputs that are more likely to trigger vulnerabilities. Our evaluation shows that our approach to fuzzing smart contracts out-performs all current comparable smart contract analysis approaches, i.e., it outperforms current fuzzers and symbolic executors. Furthermore, this chapter describes how to accurately handle the complex interactions possible on the Ethereum blockchain in a fuzzer: cross-contract interactions, multiple colluding attacker accounts, and various types of reentrant calls. Our high-throughput fuzzer is paired with a very precise bug oracle based on the native cryptocurrency of Ethereum. This means that any issue identified by our fuzzer is very likely an exploitable bug. In fact, the fuzzer presented in this chapter automatically generates exploits that can be used to verify the presence of the vulnerability.

The contents of this chapter are based on the following paper:
"EF/CF: A High Performance Fuzzer for Ethereum Smart Contracts". ***IEEE European Symposium on Security and Privacy (EuroS&P)*, 2023**. Michael Rodler, David Paaßen, Wenting Li, Lukas Bernhard, Thorsten Holz, Ghassan O. Karame, and Lucas Davi

Exploit Mitigation for Smart Contracts using Taint Tracking Chapter 6 describes a runtime exploit mitigation technique for smart contracts, which is called SEREUM. In this dissertation, we develop a novel dynamic analysis technique that is capable of identifying reentrancy attacks, which are responsible for several high-profile attacks on the Ethereum blockchain. In contrast to prior work, this approach focuses on detecting attacks as they are executed. SEREUM extends the Ethereum virtual machine (EVM) with a taint tracking engine, allowing us to track information about data flows at runtime. In turn, SEREUM utilizes the taint tracking engine to implement an automatic locking mechanism in the EVM runtime environment to detect abuses of reentrancy bugs. An attempted attack can be mitigated based on SEREUM's accurate attack detection: Our approach can be embedded into the EVM execution environment, such that reentrancy attacks are blocked automatically. Alternatively, our approach can be used by smart contract developers to monitor smart contract executions for reentrancy attacks and quickly react to ongoing attacks.

This chapter is based on the following publication:
"Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks". ***26th Annual Network and Distributed System Security Symposium (NDSS)*, 2019**. Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi

Automatic Patching of Smart Contracts with Bytecode Rewriting Finally, this dissertation discusses patching of smart contracts in Chapter 7. Due to the nature of smart contracts, which are immutable after deployment and always available, patching is highly challenging. Prior to the work presented in this chapter, patching smart contracts relied on an error-prone manual process.

To reduce the potential for errors, Chapter 7 presents a framework for automatic patching of smart contracts: EVMPATCH. At the core of this framework is a trampoline-based bytecode rewriting approach that rewrites and patches EVM bytecode. To deploy the patched contract, EVMPATCH utilizes a feature of the EVM that allows the reuse of code from another smart contract to circumvent the immutability. This chapter presents an evaluation of the feasibility of the EVMPATCH approach by integrating an existing vulnerability detection component to automatically patch integer bugs in smart contracts. Developers can use our framework to quickly react to smart contract vulnerabilities and deploy patched smart contracts.

In this chapter, methodology and results are presented that are based on the following publication:
"EVMPatch: Timely and Automated Patching of Ethereum Smart Contracts". ***30th USENIX Security Symposium*, 2021**. Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi

The results of this dissertation enable software security improvements for TEEs and smart contracts. TEEREX (Chapter 4) and EF$\frac{1}{2}$CF (Chapter 5) enable developers to automatically identify security vulnerabilities before deployment of software into secure execution environments. With SEREUM (Chapter 6) this dissertation presents an extension to the execution environment to detect attacks at runtime. Finally, with EVMPATCH (Chapter 7) this dissertation presents an approach to patching in always online secure execution environments. Therefore, this dissertation covers security across the full development lifecycle of software for secure execution environments.

Additional Publications

Apart from the main publications on which this dissertation is based on, the author of this dissertation was involved in several research projects and co-authored the following publications:

- *"Control Behavior Integrity for Distributed Cyber-Physical Systems"*. ***11. ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2020**. Sridhar Adepu, Ferdinand Brasser, Luis Garcia, Michael Rodler, Lucas Davi, Ahmad-Reza Sadeghi, and Saman A. Zonouz
- *"My Fuzzer Beats Them All! Developing a Framework for Fair Evaluation and Comparison of Fuzzers"*. ***26th European Symposium on Research in Computer Security (ESORICS)*, 2021**. David Paaßen, Sebastian Surminski, Michael Rodler, and Lucas Davi
- *"xTag: Mitigating Use-After-Free Vulnerabilities via Software-Based Pointer Tagging on Intel x86-64"*. ***IEEE European Symposium on Security and Privacy (EuroS&P)*, 2022**. Lukas Bernhard, Michael Rodler, Thorsten Holz, and Lucas Davi
- *"Practical Mitigation of Smart Contract Bugs"*. **arXiv: `2203.00364`, 2022, (under submission)**. Jens-Rene Giesen, Sebastian Andreina, Michael Rodler, Ghassan O. Karame, and Lucas Davi
- *"Dissecting and Fuzzing Native Code on the Web"* **(under submission), 2023**. Oussama Draissi, Tobias Cloosters, David Klein, Marius Musch, Michael Rodler, Lucas Davi, and Martin Johns

CHAPTER 2

BACKGROUND ON SOFTWARE VULNERABILITIES

Software vulnerabilities are a major threat to modern IT systems, as software defines most of the general-purpose computing devices we use today. Furthermore, software systems have become highly connected, which exposes the software to potential attacks. In this chapter, we cover the necessary technical background on software vulnerabilities and standard techniques to automatically identify vulnerabilities. Here this chapter focuses on automatic testing.

Many vulnerabilities in software stem from programming mistakes that can be categorized into various classes of bugs. For example, memory corruption exploits are a major threat to systems software. Most systems software is written in the memory-unsafe programming languages C/C++, which gives attackers the possibility to launch especially devastating attacks once a memory-safety bug is discovered. We introduce memory safety and *memory errors* in Section 2.1.1. However, memory safety is far from being the only type of vulnerability. We also discuss other relevant classes of vulnerabilities, such as *concurrency bugs*, in Section 2.1.2. Finally, we also discuss security vulnerabilities due to mistakes in the program's business logic Section 2.1.3.

Automatically identifying bugs in software, both from a quality and security viewpoint, is a large field of research. Multiple analysis techniques have been proposed, ranging from static analysis methods to automated dynamic testing of code. Like software libraries, secure execution environments expose a public interface, colloquially known as an application programming interface (API). We give an overview of the most important related work in the area of automatic testing of APIs, as these works are relevant also to automated testing of the APIs of secure execution environments, such as enclaves or smart contracts. More specifically, Section 2.2 covers two state-of-the-art techniques in automatically generating inputs for software APIs: symbolic execution (Section 2.2.1) and fuzz testing (Section 2.2.2). We utilize symbolic execution to analyze SGX enclaves in Chapter 4 and fuzz testing to generate smart contract exploits in Chapters 4 and 5.

2.1 Vulnerabilities in Software

This section reviews the most important classes of software vulnerabilities that are relevant in later chapters. We describe the necessary background on memory safety issues in Section 2.1.1, which have been a significant issue for large C/C++ code bases for almost 30 years [Sze+13]. In Chapter 4, we describe memory corruption attacks that exploit the lack of memory safety at the boundary between trusted SGX enclaves and the untrusted host application. Furthermore, we discuss vulnerabilities that are due to issues with concurrent execution in Section 2.1.2, which is relevant for some of the exploits we describe in Chapter 4 and reentrancy attacks in smart contracts, as discussed in Chapters 5 and 6. Finally, we also discuss security vulnerabilities due to logic bugs in Section 2.1.3, which is highly relevant in the context of smart contracts (Chapters 5 to 7).

2.1.1 Memory Safety Violations

Even after more than 20 years of research on memory corruption bugs [One96] and defenses [Cow+98] in C/C++ applications, this problem has still not been completely addressed [Sze+13]. While memory-safe systems programming languages, such as Rust or Go, have become more popular, many legacy codebases are still written in unsafe C/C++. One of the most critical and prominent examples is web browsers [Lim+21]. Today web browsers have become so feature-rich and ubiquitous that most computer users will regularly use their web browser to safely execute code loaded from potentially untrusted sources. Modern websites have evolved into web apps that consist of a mixture of HTML, CSS, multimedia content, and most critically, JavaScript and WebAssembly code. This gives an attacker a huge attack surface, while the user relies on the web browser to securely execute the web app and restrict any illegitimate data accesses [Jia+16; Lim+21]. While browser vendors employ significant effort to identify and mitigate memory errors, web browsers are still regularly exploited using security-relevant memory errors. Similarly, operating system components are still mostly written in C/C++ and are critical to the overall system's security. As such, finding new techniques to protect legacy C/C++ code against memory corruption attacks remains crucial.

While memory errors are still prevalent, they have become significantly harder to exploit in practice. Multiple exploit mitigation techniques have been developed and deployed to production systems. Hardware-enforced non-executable memory essentially stopped code injection attacks [PaX], and stack canaries made exploitation of stack-based buffer overflows less likely to succeed [Cow+98]. Control-flow integrity (CFI) [Aba+09] is a well-researched mitigation that is now widely available in production compilers [Mic14] and prevents return-oriented programming (ROP) [Sha07] attacks. However, these mitigations have forced attackers to resort to more advanced code-reuse or data-oriented attacks [Hu+16; Isp+18; PKH19; Sch+15b].

Memory safety can be roughly categorized into two dimensions: *spatial* and *temporal* memory safety. Spatial memory safety concerns itself with indexing with respect to a base pointer, i.e., there should be no out-of-bounds memory access due to pointer arithmetic. The most prominent example of a spatial memory error is the *buffer*

overflow, where the program writes past the end of an allocated buffer. Temporal memory safety is concerned with the lifetime of a memory object, i.e., there should be no usage of a memory object before it was allocated or after it was deallocated again. An example of a bug that violates spatial memory safety is the *use after free* bug. Here a pointer to a deallocated object remains in memory and the program accesses the already deallocated object through this dangling pointer. The memory model exposed to a C/C++ programmer is very low-level. As a consequence, these languages provide neither temporal nor spatial memory safety. Low-level access to memory is necessary to implement software components that work directly with hardware or to develop extremely efficient optimized software. Most bug classes can be classified as either a spatial or temporal memory safety violation. However, orthogonal but also closely related to the problem of memory safety is the weak type safety offered in C/C++. We discuss the relationship between type safety and memory safety at the end of this section.

Spatial Memory Safety To violate spatial memory safety, two conditions must be met: 1) a pointer must be adjusted to point outside the bounds of the original object, and 2) the pointer must be dereferenced. Here, the most common root cause of spatial memory safety violations is pointer arithmetic and a lack of proper bounds checking. The buffer overflow is the classic example, where due to lack of length checking, the program accesses an array, or buffer, past the allocated length. However, also other bugs can be classified as spatial memory errors. For example, if a program reads dereferences an uninitialized pointer, then the program will attempt to access an object somewhere in memory. In contrast to issues with pointer arithmetic, there is no associated base pointer. However, the pointer may be partially under attacker control, leading to a security vulnerability beyond a simple crash. Other bugs that lead to spatial memory safety violations are type casts between incompatible types (type-confusion) and dereferencing of invalid pointers, such as uninitialized or NULL pointers.

Figure 2.1 shows an example of a spatial memory safety violation in the C programming language. Here arrays are represented as pointers to the first object in the array. In this case, the first byte of the *char* array. The array subscript operator is simply syntactic sugar for pointer arithmetic: it is translated into a pointer addition and a pointer dereference. The *for* loop in the example will execute the loop body nine times due to the $\leq$ operator being used in the loop header. As a result, the buffer index `i` will index the ninth element in the last iteration, resulting in a buffer overflow that writes the bounds of the original `array` object.

Such buffer overflows easily occur when dealing with strings or byte buffers in the C language. This is mostly due to the absence of automatic bounds checking and the poorly designed string APIs of the C standard library [One96]. While string handling in C++ is much less error-prone due to the *STL* `std::string` type, other common container types still default to unchecked operations. For example, the common `std::vector` container defaults to using unchecked access via the subscript operator while providing bounds-checked access only via the `at` method. Memory-safe languages, such as Java, go or rust, deal with the problem of spatial memory safety

```
1  char array[8];
2  for (int i = 0; i <= 8; i++) {  // off-by-one error
3    array[i] = '\0';  // last loop will set array[8]
4  }
```

Figure 2.1: Example of a spatial memory safety violation (buffer overflow) due to an off-by-one error, where the last loop iteration executes with a buffer index that is too big.

by automatically introducing bounds-checking code, disallowing arbitrary pointer arithmetic, and preventing dereferencing of uninitialized pointers.

Temporal Memory Safety A program is temporal memory-safe if no objects are accessed before or after they have been allocated. A program must first let a pointer become *dangling*, i.e., a pointer that references an object that has been deallocated. If this dangling pointer is dereferenced, temporal memory safety is violated. Temporal memory safety violations become especially problematic if the memory area that the dangling pointer is referencing contains another potentially attacker-controlled object because memory is reused by the allocator. Temporal safety violations are often due to the manual memory management in C/C++: the programmer has to explicitly allocate and free memory. Even moderately large code bases, it is not easily possible to track all references to an object and decide when it can be freed.

Figure 2.2 shows an example of a temporal safety violation. First, 8 byte of memory are allocated using the standard *malloc* function in line 3. A reference into the allocated memory is saved in the pointer `q`. Subsequently, the allocated memory is again deallocated using the standard *free* function. Afterward, another 8 byte memory object is allocated. To reduce memory usage, a typical implementation of a memory allocator will attempt to reuse the previously freed memory block and immediately return it to the caller. As a result, when the pointer `q` is dereferenced in the last line, it would actually write to the `u` object instead. However, at this point, both pointers alias each other, even though they should not.

```
1  uint32_t *p, *q;
2  char *u;
3  p = malloc(8); // allocate a uint32_t[2] array
4  q = p + 1;  // q references the second uint32_t
5  // ...
6  free(p);
7  u = malloc(8); // likely(p == u)
8  // ...
9  *q = ...  // Use-After-Free Bug: modifies u instead
```

Figure 2.2: Example of a temporal memory safety violation.

One can utilize automatic memory management, typically by employing garbage collection or reference counting, to automatically and comprehensively ensure temporal

memory safety. This way, a memory location is only deallocated when no reference exists anymore. Garbage collection has been popularized by languages such as Java or C#, which feature tracing garbage collectors. However, C/C++ can also be retrofitted with garbage collection [BD95], and most modern systems programming languages also feature automatic memory management. For example, the *go* language has implemented an efficient and lightweight garbage collector. The Rust programming language features the borrow checker to prove temporal memory safety at compile time at the expense of becoming more restrictive with respect to allocation and aliasing behavior [Jun+20]. A Rust programmer can fall back to reference counted smart pointers when the compiler cannot prove temporal memory safety at compile time. The C++ *STL* has also introduced smart pointers, which automatically manage the lifetime of the underlying object. However, due to backward compatibility, C++ smart pointers are often turned into unmanaged legacy pointers, bypassing the guarantees of the smart pointers. As such, temporal memory safety is still a big issue for large C++ code bases such as browsers, where *use-after-free* vulnerabilities are still the majority of vulnerabilities [Zer].

Type Safety vs. Memory Safety Type systems are the foundation of many programming languages. At the most basic level, types are used to ensure that the programmer attempts to combine only compatible objects in operations, e.g., only numbers are added and not a number and a string. However, type systems also facilitate abstractions and interfaces, allowing the programmer to hide complexity, reuse code, etc. If a compiler's type checker cannot decide whether two types are compatible, the decision can be postponed to the runtime of the program. This is colloquially referred to as dynamic typing. However, the C language has no concept of types during runtime. All type checking happens during compilation (static typing), and at runtime, the programmer has to ensure that only compatible types are used. However, type casting and the union data structure provide facilities to convert any type to any other type, allowing the programmer to break any guarantees by the type system.

Furthermore, in C and C++, type safety is also closely intertwined with memory safety. The lack of type safety can be easily turned into a memory safety issue when a type cast wrongly changes the meaning of an integer to a pointer, leading to out-of-bounds access. Inversely, for type safety to hold during runtime, memory safety must also hold. A memory safety violation can lead to breaking a type invariant, e.g., a boolean value that is represented by neither 0 nor 1 in memory.

The weak type system can also lead to memory safety violations. Figure 2.3 shows an example where the lack of proper type-checking results in an out-of-bounds access. The class `Bar` inherits from `Foo`, so it has the members `a` and `b`. A pointer to a `Foo` is cast in the code snippet to the `Bar` type without additional checks. However, in this case, the pointer actually references a `Foo` object. This type of object does not feature the `b` member, which is accessed later. This access leads to indexing out-of-bounds of the original object, whose size is only 4 byte.

The C language features a notoriously weak type system, which causes many issues and is often mitigated using ad-hoc object models. As such, the C++ object model

```
1  struct Foo {
2    int a;
3  }; // sizeof(Foo) == 4
4  struct Bar : public Foo {
5    int b;
6  }; // sizeof(Bar) == 8
7  // ...
8  Foo* f = new Foo();
9  // ...
10 Bar* b = static_cast<Bar*>(f);  // invalid downcast
11 b->a = ... // *(b + 0) ok
12 b->b = ... // *(b + 4) out-of-bounds
```

Figure 2.3: Weak type safety in C/C++ can cause memory safety issues.

requires that for certain classes, the type of an object can be acquired at runtime using *runtime type information* (RTTI). This information can then be used to validate type casts using the `dynamic_cast` operator. Unfortunately, this does not mitigate all typing-related issues in C++. Furthermore, RTTI is often disabled for performance reasons [Jeo+17].

2.1.2 Concurrency Bugs

Since current software systems typically run multiple tasks on one machine, they need to support concurrent execution of programs. However, concurrency in software systems complicates software design and implementation. There are several bugs that are easy to introduce when writing concurrent software. We will now discuss the major concepts of bugs in concurrent software systems.

Race Conditions Adve et al. [Adv+91] defines races conditions in the following way:

> Informally, two memory operations in an execution form a data race if at least one of them is a data operation (as opposed to a synchronization operation), at least one of them is a write (as opposed to a read), both access the same memory location, and they are not ordered by intervening synchronization operations.

As such, race conditions often lead to problems, such as two threads operating on different values, causing corruption of the values. In the broader sense, race conditions are a problem for software robustness. However, race conditions can also become the root cause of memory corruption bugs. For example, there have been multiple race conditions in the Linux kernel that cause use-after-free (UAF) bugs [LML21].

Time-of-Check to Time-of-Use Issues The problem behind time-of-check time-of-use (TOCTOU) issues is that there is a time window between a security check and the use of a resource. For example, TOCTOU bugs are common when operating on files. For example, a privileged program or service validates that an input file is not

```
1  // An argument marked as "__user *" is a pointer to the user space memory passed
2  // to the kernel by the user space code.
3  void tls_setsockopt_simplified(char __user *arg) {
4    struct tls_crypto_info header, *full = /* allocated before */;
5
6    // [FETCH]
7  ①  if (copy_from_user(&header, arg, sizeof(struct tls_crypto_info)))
8      return -EFAULT;
9
10   // [CHECK]
11 ②  if (header.version != TLS_1_2_VERSION)
12     return -ENOTSUPP;
13
14   // [FETCH]
15 ②  if (copy_from_user(full, arg, sizeof(struct tls12_crypto_info_aes_gcm_128)))
16     return -EFAULT;
17
18   // [BUG] full->version might not be TLS_1_2_VERSION
19   // ...
20   // process(full)
21   // ...
22 }
```

Figure 2.4: Example for a double-fetch bug in the networking code of the Linux kernel [Xu+18].

a symbolic link before accessing the file. This is to avoid confused deputy attacks, where the attacker abuses a privileged program or service to access a file normally out-of-scope for the attacker. However, the attacker can attempt to exploit the time window between the check on the file, e.g., by replacing the file with a symbolic link after the check has passed.

Double-Fetch Bugs This class of bugs has become infamous due to the prevalence in operating system (OS) kernels with support for multi-threaded user space processes. Here the kernel fetches the same user space memory location two times in a row with the assumption that the value cannot change between the two reads. However, in a multi-threaded system, this is not necessarily the case. A typical attack vector is the execution of a system call. The kernel fetches a system call parameter from user space, performs some validation, and then fetches the same parameter again from user space memory. However, if the user space process has the capability of running multiple threads, it can change the value between the first and second fetch. This often leads to TOCTOU issues, where the user space can bypass kernel validation by changing memory parameters after the validation is finished.

Figure 2.4 shows an example of a double-fetch bug that causes a potential TOCTOU issue. Here the kernel attempts to validate whether the user space program passed the right version of a TLS header to the kernel. It does so by checking the TLS version flag in the header data structure. First, it fetches a general header from user space at ①. It then performs a check for the TLS version by comparing it with a predefined header

(②). The kernel then fetches the full-sized header from user space at ③. However, in between ① and ③, the user space can change the value of the header to something else.

In general, the difficulty of exploiting race conditions varies depending on the target system and is influenced by many factors, such as the number of running processes, general system load, the microarchitecture of the CPU, etc. For example, the window between the two memory fetches in Figure 2.4 seems very small: only a single comparison and a conditional branch. However, an attacker often has various ways to extend such *race windows* when attempting to exploit race conditions. For example, the attacker can exploit microarchitectural features, such as caches or the branch predictor, to stretch the race window [Sch+18]. Other techniques abuse some kernel functionalities to extend a race window, such as moving threads between CPUs or triggering interrupts [LML21].

2.1.3 Logic Bugs

In previous sections, we discuss security vulnerabilities that are due to shortcomings of the used programming language or environment. For example, memory errors are mostly due to the lack of built-in memory safety in systems languages such as C/C++. However, most application languages, such as Java, offer memory safety using automatic bounds checking and garbage collection. However, security vulnerabilities can also arise due to programming mistakes in an application's logic. These are independent of the programming language and typically highly specific to the domain of the application. For example, many systems have access control rules to enforce integrity and confidentiality of the data on the system. However, a bug in the enforcement logic of the access control check can result in a security hole that attackers can exploit to work around the usual access control checks. However, the class of logic bugs contains different application-specific issues going far beyond access control issues. Broadly speaking, the class of logic bugs contains all vulnerabilities that allow an attacker to manipulate and abuse the legitimate processing flow of an application with a negative impact on the organization running the application or to other users of the application.

Figure 2.5 shows an example for a logic bug. The depicted function is part of a larger application that manages transactions that transfer some form of currency between users, for example, the in-game currency in a multiplayer game. The function receives two user IDs and the amount, represented as a signed integer, as parameters. The code includes a check for whether the original user has enough currency in their account. However, the code does not account for the fact that a malicious user could attempt to transfer a negative amount. As a result, the malicious user can abuse the transfer of negative amounts to force the system to transfer currency from other accounts without any permission or checks. In this example, this violates the application's intended logic: no user should be able to steal currency from other accounts.

```
// ...
int get_user_balance(userid_t user);
void set_user_balance(userid_t user, int amount);

void transfer(userid_t from, userid_t to, int amount) {
  if (get_user_balance(from) >= amount) {
    int new_balance_from = get_user_balance(from) - amount;
    set_user_balance(from, new_balance_from);
    int new_balance_to = get_user_balance(to) + amount;
    set_user_balance(to, new_balance_to);
  }
}
// ...
```

Figure 2.5: The depicted function manages bank transfers, sending some amount from one user to another. However, the application logic does not account for negative amounts leading to a vulnerability, where anyone can steal money from any other account.

2.2 Testing Software Interfaces (APIs)

Developers generally decompose Software into various independent pieces, which are then combined to a full software product. For example, it is a common model to encapsulate commonly useful functionality into software libraries. A single program then consists of the main program code and associated libraries. This has the advantage of allowing the re-use of library code, which increases maintainability of software, allows for distributed development, and also increases the resource efficiency of software. Similarly, distributed software systems are composed of multiple software services, which interact with each other over the network. Every library or service exposes a public interface, which users of the library or service can call to use the functionality implemented in the library. This interface is commonly referred to as the API.

Generally, there is no trust boundary between a main program and the libraries it utilizes. However, when considering services in distributed systems, we often observe a trust boundary between services that potentially run on different machines. For example, many web services offer public APIs for anyone to interact with. Such APIs often include access control logic to authenticate and authorize the user of the API. When considering secure execution environments, we also deal with a certain type of API. For example, in the case of SGX, an enclave acts as an isolated software library that can be utilized by an untrusted main program. Similarly, smart contracts act as services, which can be called by everyone that takes part in the Blockchain network. As such, similar to, e.g., web services, software in secure execution environments must perform self-protection: it must ensure that the encapsulation is not violated due to errors on the interface trust boundary, and it must perform access control (authentication and authorization).

Since the API of any software component is the entry point to its functionality, it must be well-tested. This is especially true for software where the API represents a trust boundary, such as for software running in secure execution environments. Testing

APIs is a highly challenging problem because most APIs are stateful. This means that prior calls can influence the behavior of following calls. When testing stateful APIs, one has two consider two dimensions: 1) The individual inputs to a single API call, 2) and the ordering of the call sequence, taking interdependencies into account. Both dimensions already represent a considerable search space on their own. This makes testing of stateful APIs a long-standing open research topic [Ait02; Bal+18; CH00; KLT01; Man+21; Thu+11; Xie+05].

When it comes to automated test input generation, there are two major techniques that have been developed over the years: symbolic execution [Bal+18] and fuzzing [Man+21]. Both techniques received significant attention from the research community and have had tremendous success in identifying bugs and security vulnerabilities [Cha+12; Zal]. In the remainder of this section, we discuss both techniques in the context of automatically testing software APIs.

2.2.1 Symbolic Execution

Symbolic execution has become one of the standard techniques in software analysis, including high coverage testing and vulnerability analysis [Bal+18; CDE08; Cha+12; Sho+16]. While on in the 70s as a generalization of testing [BEL75; Kin76], it gained traction only recently with advances in satisfiability modulo theories (SMT) solving [BT18; MB08] and high-performance SMT solvers being widely available [Bar+11; Dut14; MB08]. The idea behind symbolic execution is to generalize testing by executing the program within a symbolic domain instead of concrete values, e.g., the program operates on *symbols* representing numbers instead of actual numbers. This allows the symbolic execution to reason about classes of inputs at once instead of executing tests one by one. In turn, the symbolic execution discovers complex edge cases that often do not come up during normal testing.

In practice, this is done by replacing variable inputs with placeholders, fresh symbolic values, at the start of the execution of a program. The program then executes with the symbolic values as inputs. For example, a fresh symbolic value for a 32 bit integer represents the set of all possible integer values. Assignments to other variables propagate the symbolic values to other variables during symbolic execution. Arithmetic expressions result in symbolic formulas over the involved symbolic values and constants. Conditional expressions are especially challenging during symbolic execution. At every branching point in the program, the symbolic execution engine must decide which branch to take. It does so by checking the satisfiability of the path constraints π, representing all constraints on the input symbols gathered. Taking a branch adds another constraint to the set of path constraints. If both branches are feasible, the symbolic execution engine must fork the execution and take both branches.

Figure 2.6 shows an example of the state tree spanned by the symbolic execution of the depicted function *foobar*. In this example, the symbolic execution branches into both directions at each of the if-conditions.

As we saw in the example in Figure 2.6, symbolic execution spans a tree over all possible execution paths of the program. In small programs such as the example in Figure 2.6, a full exploration of all paths is feasible. However, in large real-world programs, the execution tree quickly becomes too large to explore fully. Several

```
1  void foobar(int a, int b)
2  {
3    int x = 1, y = 0;
4    if (a != 0) {
5      y = 3 + x;
6      if (b == 0) {
7        x = 2 * (a + b);
8      }
9    }
10   assert(x - y != 0);
11 }
```

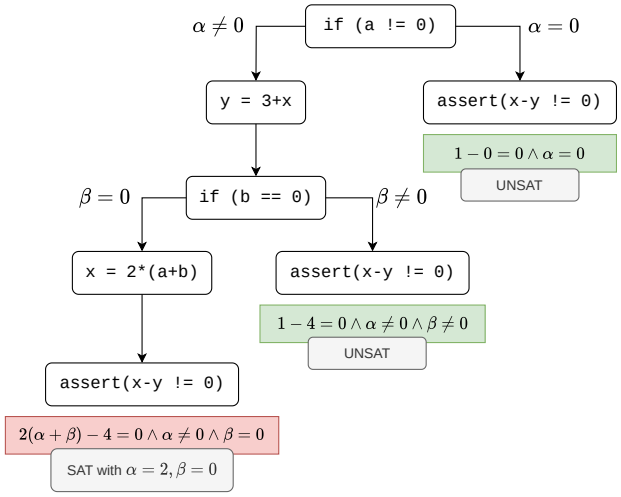


Figure 2.6: Example for an execution tree spanned by the symbolic execution of the depicted function *foobar* [Bal+18] with α and β being the symbols assigned to the input variables `a` and `b`, respectively. Only in the left-most path in the execution tree is it possible to violate the assert statement. The symbolic execution produces a satisfying assignment, i.e., concrete values that can be passed as inputs and violates the assertion.

common features of real-world code quickly introduce a *path explosion* problem: nested loops, symbolic memory dereferences, input-dependent indirect jumps, etc. Because the symbolic execution engine must keep both states in memory at every conditional jump, symbolic execution quickly becomes memory-bound, making breadth-first traversal of the execution tree infeasible. As such, modern symbolic execution engines feature a combination of depth-first traversal and heuristics to guide the symbolic execution to potentially interesting program locations. Dealing with the path explosion problem is a major research direction in symbolic execution, which spawned new techniques such as state merging, computing symbolic summaries, different memory models, and many more techniques [Avg+14; Bal+18; Kuz+12].

Furthermore, to be practically useful in real-world software, symbolic execution must accurately model side effects caused by the environment. For example, most software expects a standard OS environment, where the symbolic execution engine must simulate and support all OS system calls and manage a simulated file system [Bal+18]. To avoid state explosion, many symbolic execution engines also hook and model common library functions [Sho+16].

Concolic Execution The idea of concolic execution is to mix concrete and symbolic program execution. As such, the name of this variant of symbolic execution is a portmanteau of the words “concrete” and “symbolic”. Concolic execution is often used synonymously with *dynamic symbolic execution* and usually acts as a test-case generator [Cad+08; GKS05; GLM12]. In general, most concolic execution engines will execute a concrete input that drives the symbolic analysis along a certain program path. Symbolic values and path constraints are tracked alongside the concrete inputs

as meta-data. The concolic execution engine can then utilize the gathered path constraints to flip one or more branches along the concrete execution.

One of the major advantages of concolic execution is that the environment must not be modeled explicitly. Instead, since the execution is driven by the concrete execution, the calls into the environment simply execute normally. Of course, this comes at the expense of not being able to determine faults that are due to unexpected data returned by the environment.

The many scalability problems of symbolic execution make it harder to apply standard static symbolic execution to modern software. As such, most symbolic execution approaches trade soundness or completeness of the analysis for shorter analysis times. Nevertheless, symbolic execution has been successfully applied to identifying bugs and security vulnerabilities in a wide range of software [CDE08; CKC11; Sho+16].

2.2.2 Fuzz Testing

Over the last years, fuzzing has become a huge research topic in both the software engineering and security communities [Man+21]. Driven by the practical usefulness and success in identifying a large number of impactful bugs, fuzzing has been adapted to many types of software. Historically, there have been two major branches of fuzzing approaches: generative and mutational. Generative approaches utilize an input specification to probabilistically synthesize inputs that conform to the specification. Mutational approaches start with a set of seed inputs and perform small mutations, slightly changing the input.

Generative Fuzzing Generative fuzzing utilizes an input specification, sometimes called a model, to generate new inputs from the specification. The most prominent class of fuzzers in this area is grammar fuzzing. Grammar fuzzers are especially well suited to test parsing code where a formal grammar is already available. A formal grammar can be used to decide whether a sentence is valid, but it can also produce a valid sentence in the language described by the grammar. As such, it is only natural to turn the grammar used by a parser around and utilize them to test the parser. A grammar fuzzer then performs random sampling on the grammar to generate valid inputs. Typically, the grammar used by the fuzzer slightly deviates from the grammar used by the parser to make the fuzzing more efficient and more likely to trigger bugs [Asc+19a]. As such, grammar fuzzing has been successfully applied to a wide range of software, such as programming language compilers [Asc+19a; HHZ12; Yan+11], standardized serialization formats, or even testing a single function [CH00]. The biggest downside is that a precise specification of the input must be available. While inferring input grammars from static and dynamic program features has been researched in the past [Bas+17; GMZ20; HZ16; KLS21], a different approach to fuzzing has proven more effective for scenarios where no input specification is available.

Mutational Fuzzing Mutational fuzzing has become the de-facto standard approach to testing binary-format parsers. Here, the fuzzer takes a set of known valid input files, the seed corpus, and performs small-scale mutations, such as bit-flips or character

replacements, to produce new inputs. Already a blind mutational approach can identify a large set of bugs [RDMSA]. The results of a mutational fuzzer highly depend on the quality of the seed corpus and on the capability of the fuzzer to quickly execute many of the generated inputs. While mutational fuzzers are well suited to test the robustness of parsers, they tend to uncover only *shallow* bugs, i.e., they mostly exercise the error handling code of the parsing components.

However, mutational fuzzing becomes exceptionally effective when combined with *feedback* on the execution of the target program. For example, the famous fuzzer *AFL* [Zal] utilizes code coverage feedback to distinguish inputs. For every generated test case, the fuzzer receives feedback about the code coverage that the test case triggered in the target program. This allows the fuzzer to distinguish and classify inputs according to the code reached in the target. Fuzzers, such as AFL or *libfuzzer*, utilize this mechanism to incrementally extend the seed corpus with new interesting inputs, i.e., those that trigger new code coverage. Furthermore, this allows the fuzzer to shrink individual inputs to discard bytes that do not affect which code executes. Similarly, the fuzzer can now shrink a whole corpus, removing unnecessary test cases that trigger only code already covered by other test cases in the corpus. Since the feedback-driven mutational fuzzer can now incrementally build a corpus of interesting inputs, the fuzzer can now reach code paths using small-scale mutations. For example, Zalewski [Zal14] demonstrated that AFL is capable of synthesizing complex file formats, such as *JPEG* images, given only code-coverage information of a *JPEG* parsing library.

More recently, several fuzzers were proposed that combine aspects of mutational and generative fuzzing. For one, many generational fuzzers adopted coverage feedback to distinguish useful inputs. Furthermore, this allows one to extend the concept of mutations to the grammar level: the fuzzer now parses, mutates, and emits the parse tree of the target grammar as source code. This was shown to be more effective than traditional purely generative grammar-based fuzzers [Asc+19a; Wan+19b]. On the other hand, mutational fuzzers have also approached generational fuzzers by discovering input specifications driven by the mutational fuzzing process [Bla+19].

Hybrid Fuzzing A hybrid fuzzer augments a coverage-guided fuzzing approach with more involved program analysis techniques, such as symbolic execution or taint tracking. This allows the analysis to explore large parts of the program using lightweight coverage-guided fuzzing and only resort to heavy-weight analyses whenever the fuzzer fails to make further progress.

Even though mutational fuzzers receive feedback about code coverage, they lack any semantic understanding of the input bytes they mutate. For example, such a fuzzer cannot determine which bytes need to be mutated to discover new code coverage. Using taint analysis, one can determine the relationship between input data and data processed in the program. For example, by performing taint analysis, a certain comparison can be traced back to the respective bytes in the input bytes. The fuzzer can then focus on mutating those bytes that influence a certain branching condition instead of randomly choosing the input bytes to mutate [Bek+12; CC18; Raw+17].

While taint tracking gives the fuzzer the information on what part of the input to mutate, it does not give information on how to mutate the input. In contrast, symbolic

execution techniques gather path constraints and can generate inputs that satisfy even complex constraints on the input. A notable example here is *Driller* [Ste+16], which combines coverage-guided fuzzing based on *AFL* [Zal] with concolic execution with the *Angr* framework [Sho+16]. Conceptually, Driller splits code into regions called *compartments*. To determine a compartment, Driller first executes the fuzzing component. A compartment is defined as the code that is exercised by the fuzzer during the first phase. Once the fuzzer fails to make further progress in terms of code coverage for a certain amount of analysis time, Driller invokes the concolic execution engine to discover new compartments. It does so by solving complex path constraints gathered along the paths already exercised by the fuzzer and then flips those branches that lead to code outside of the current compartment, i.e., those that trigger new code coverage in the fuzzer. Driller then repeatedly cycles between the fuzzing and concolic execution components.

Somewhat counter-intuitively, in terms of identifying real bugs in complex code bases, smarter mutation strategies such as those based on heavyweight program analysis techniques [GLM12; Raw+17] are often outperformed by more lightweight approximative mutations that can be applied at a high throughput [Asc+19b; Cho+19; Fio+20]. However, most recently, advances in concolic execution have been proposed to speed up hybrid fuzzing. For example, one can exploit the fact that a concolic executor is most commonly used in conjunction with a coverage-guided fuzzer. This allows the concolic executor to simplify the symbolic formulas in an unsound manner and offload the task of verifying the input to the coverage-guided fuzzer [Yun+18]. Furthermore, the overhead of running the program in a fully emulated interpreted symbolic environment [Sho+16] can be reduced by instrumenting directly on the assembler level [Yun+18], instrumenting the program for concolic execution in the compiler [PF20], or in the just-in-time (JIT) translation layer of dynamic binary translation tools [BCD21b; PF21]. As an additional optimization, approximative solving optimized for constraints gathered during concolic execution has been proposed [BCD21a; Che+22] to replace expensive queries to SMT solvers.

Fuzzing Stateful APIs The goal of fuzzing is to generate a sequence of API calls, where (1) the final call in the sequence will trigger a bug, (2) while the preceding calls set up the state of the target such that the bug can be triggered. This results in a very large search space for the analysis tool, as one must consider potentially infinite call chains. Even if the length of a call chain is bounded to a reasonable number during fuzzing, the fuzzer must still search along the two different directions of individual function input and function ordering. However, coverage-guided mutational fuzzing allows the fuzzer to effectively search this huge search space. Most of the function sequences will be redundant and not lead to distinct code coverage, allowing the fuzzer to quickly discard these generated call chains. However, code coverage is not always sufficient to distinguish the inputs, as different interesting combinations often do not lead to different coverage. As such, a fuzzer must carefully strike a balance between incorporating additional coverage information, such as context-sensitive code coverage [CC18; Fio+20], and unnecessarily increasing the size of the current corpus.

Additionally, the fuzzer often has to take interdependencies between the calls into account: data dependencies from return values of prior calls to parameters of following calls or control dependencies between calls. Data dependencies often occur when the API uses special identifiers to refer to resources. For example, the Linux kernel extensively uses file descriptors at the system call interface. Passing invalid file descriptors is very likely when considering it as an integer. So the fuzzer must ensure to use only those file descriptors, which the kernel has previously returned. Similarly, control dependencies encode semantic relationships between calls. For example, in the case of file descriptors, the fuzzer should avoid using file descriptors that have been closed already. Similarly, when fuzzing an object-oriented API, the fuzzer should not utilize objects that have not been constructed or initialized.

However, fuzzing APIs has several advantages in contrast to static analysis methods. Fuzzing has the advantage that it features a very low rate of false alarms. Since the fuzzer produces a call sequence that sets up a vulnerable state during fuzzing, it is highly likely the same call sequence also moves a real deployment into a vulnerable state. Furthermore, the generated API call sequence can be easily replayed and debugged by an analyst or developer.

In the following we describe the taxonomy as introduced by Green et al. [GA22]. This taxonomy categorizes fuzzers into one of four different types of API fuzzing approaches:

Manual Harness Coverage-guided fuzzing has been previously used to test interactive APIs. To work around the fact that most fuzzers simply provide and mutate one input byte buffer, a special fuzzing harness must be constructed. Usually, the harness performs additional parsing of the fuzzer-provided byte buffer and deserializes a sequence of API calls, i.e., the harness acts as an ad-hoc interpreter for the raw bytes provided by the fuzzer. Several libraries have been created to easily implement structured fuzzing based on the mutated byte buffer inputs provided by current fuzzers. Two examples for such libraries are the *FuzzedDataProvider* of the *libfuzzer* project [LLV] and the *arbitrary* library for rust projects [Dev22] The downside of this approach is that even though the aforementioned libraries reduce the manual effort, it still requires significant effort to create a well-working fuzzing harness. Additionally, some mutations on call sequences or constraints between calls cannot be easily expressed with this approach, as the fuzzer itself is usually unaware of the structure of the input. This can result in the fuzzer unnecessarily and repeatedly exercising the same error-handling branches.

Harness Generation Since manual harnessing requires significant effort, several research projects proposed methods to automatically synthesize a fuzzing harness. For example, FUDGE [Bab+19] and FuzzGen [Isp+20] mine large C/C++ code bases to extract code patterns and API usages to automatically generate a fuzzing harness. However, such learning approaches rely on the quality and availability of the code that is mined. Other approaches use dynamic API tracing to detect legitimate call sequences and automatically generate a fuzzing harness [HC17; Jun+21]. However, all these approaches attempt to generate a traditional fuzzing harness using a single fixed sequence of API calls. The fuzzer is then used to

mutate the input data of the fuzzing harness. Such automated harness generation makes it significantly easier to apply fuzzing to existing software projects. However, this type of generated harness is not capable of exposing bugs that result due to unexpected combinations of API calls.

Code Generation Another approach to testing APIs is to directly synthesize the source code that exercises the APIs. This approach is usually taken when fuzzing interpreted language runtimes, where it is beneficial to test the interpreter and standard library API in conjunction. This approach has been applied to many interpreted languages [Asc+19a], with a special focus on sandboxed runtimes such as JavaScript runtimes of web browsers [Gro18; HOC19; Rud07]. Usually, such fuzzers follow a grammar-based fuzzing approach, sometimes enhanced with semantic information to avoid generating invalid API call sequences. This approach works well for interpreted languages, where the input is naturally source code and is quickly executed. However, for compiled languages such as C/C++ or rust, this approach would take too long as each test case would need to go through the slow compilation and linking stages for every test case. While a grammar-based generative approach has been used to test C compilers [Yan+11], it is not practicable for testing API sequences as the low test case throughput would decrease fuzzing effectiveness.

Dynamic One crucial observation about manual fuzzing harnesses is that they often implement ad-hoc interpreters that turn raw input bytes into structures or dispatch to the relevant API. This can be generalized to an interpreter-based fuzzing harness, which has been implemented in various projects [SYZK; GA22; Sch+21; Sch+22]. Essentially, the fuzzing harness now becomes an interpreter for a bytecode format that specifies the call sequences as a sequence of opcodes executed by the interpreter. The fuzzer now mutates the bytecode program for the interpreter and its associated data. Having a well-defined generic format means that the fuzzer can be enhanced with mutations that respect the structure of the bytecode format. This avoids unnecessarily exercising error paths in the fuzzing harness and allows for more effective mutations on the call sequence. Additionally, the bytecode format can be treated as a graph of opcodes with associated data. This allows modeling control- and data-flow constraints between API calls. For example, a file must be opened before writing to it and should not be written to after it is closed. This allows expressing semantic constraints about the API usage and further increases fuzzing efficiency as fewer invalid test cases are processed during fuzzing.

CHAPTER 3

BACKGROUND ON SECURE EXECUTION ENVIRONMENTS

In this dissertation, we focus on software security within *secure execution environments*: limited, isolated, and secure environments that execute highly critical parts of a larger software system.

> **Definition:** A *secure execution environment* is an execution environment that isolates and protects the *integrity* of both code and data of the software running within the execution environment. Additionally, the secure execution environment provides some form of attestation, facilitating secure communication with software components inside of the secure execution environment.

This definition does not include *confidentiality*. This distinguishes this definition of secure execution environments from other concepts such as confidential/trusted computing, which also provide some form of confidentiality. All modern trusted computing technologies, such as TEEs, also provide integrity protection. Therefore, *secure execution environments* are a superset, i.e., include all trusted computing technologies. However, by excluding the confidentiality requirements, our definition of *secure execution environments* also covers interesting secure execution environments, such as those featured in distributed systems (e.g., blockchains with smart contracts).

In this dissertation, we cover two specific secure execution environments. It is necessary to understand the technical details of these two secure execution environments. We discuss the technical details of the SGX, which offers a hardware-backed TEE implementation. Section 3.1 gives details on the extension to the *x86* instruction-set architecture (ISA) that makes up SGX. SGX offers a confidential and integrity-protected computing environment: in theory, an enclave developer can completely shield the enclave from the host system. This covers all data used by the enclave, which can be protected using cryptographic measures that seal and bind data to a specific enclave. While this also extends to hiding the enclave's code by providing the actual enclave logic as encrypted code that is only decrypted inside the shielded enclave environment, we later only tackle the more common use case of non-confidential code.

We continue with a discussion of the Ethereum execution environment in Section 3.2, including the EVM architecture and the particularities of interactions with and between smart contracts. In contrast to SGX, the Ethereum execution environment for smart contracts exhibits many properties which result from the execution of code as part of the consensus protocol of the underlying blockchain system. For example, Ethereum has no notion of confidentiality, as everything on the blockchain is considered public. However, the consensus protocol ensures a high degree of integrity, as the consensus protocol will eventually ensure a unified view of the execution state of a smart contract across the whole network. Additionally, a smart contract is uniquely identified using an *address* that includes a hash of the code and is unique for every deployment. Using this *address* it is trivial to perform remote attestation. Finally, we conclude the background on secure execution environments by providing a comparison between the Ethereum and SGX execution environments in Section 3.3.

3.1 Trusted Execution using Intel Software Guard Extensions

Intel introduced the SGX extensions [CD16; Hoe+13; Intel4; McK+13] for the Intel x86 architecture as a new generation of trusted computing extensions, following prior attempts at trusted computing like TPM and TXT. In contrast to prior solutions, SGX provides an even further reduction of the TCB. Essentially, the TCB contains the CPU, the firmware contained in the CPU, and the code running inside of the enclave. Notably, the TCB does neither include the kernel nor the hypervisor running on the CPU. In fact, the threat model of SGX explicitly states those to be untrusted. Using secrets securely stored in Intel hardware as a root of trust and a special quoting enclave released by Intel, each SGX enabled Intel CPU can measure SGX enclaves and produce local and remote attestation reports. These attestation reports allow an enclave to verify its own identity and integrity to remote parties.

One of the primary use cases of SGX is confidential computing in cloud scenarios. On the one hand, organizations want to leverage the flexibility of cloud computing as part of their IT systems. On the other hand, due to regulatory compliance or security risks, sensitive data cannot be stored outside the organization's infrastructure. Here, SGX offers a possible solution: The cloud computing hardware utilizes the SGX extension to load and measure an enclave provided by the data owner (i.e., the organization that wants to confidentially use cloud computing). With remote attestation, the enclave can prove to the data owner that it was correctly loaded. Furthermore, the data owner can establish a cryptographically secured channel that directly terminates inside the attested enclave. Since the enclave code is considered trustworthy, the data owner can transfer the encrypted data to the enclave running on hardware operated by the cloud provider. Even if the cloud provider, or its software stack, is compromised, the data remains confidential as long as the SGX enclave operates correctly. Using the *binding* and *sealing* mechanisms of SGX, the enclave can cryptographically ensure that data can only be accessed on a certain CPU or by certain enclaves.

3.1.1 Enclave Lifecycle

Loading and Initialization On a technical level, a SGX enclave is a memory region embedded into an existing x86 virtual address space. More specifically, every SGX enclave is attached to a *host process*, a normal user space process featuring a standard x86 virtual address space. To load an enclave, the host process asks the kernel to set up the address space for the enclave. This means that the kernel will allocate the required number of pages from a special memory area reserved for SGX enclaves called enclave page cache (EPC). The kernel updates the host process' page tables to contain the necessary mapping for the enclave. The kernel must leverage special instructions to assign pages from the EPC to an enclave. These instructions initialize the enclave pages given untrusted pages. The SGX enabled CPU will record additional metadata, such as a mapping that associates physical pages with their enclaves. This allows the CPU to ensure that no page is mapped twice into different enclaves. Contrary to normal processes, the CPU must perform several sanity checks to prevent a malicious OS kernel from attacking enclaves by abusing the paging mechanism. Once the enclave is loaded, the CPU will perform the measurement of all enclave pages and their ordering. This measurement is then later used for local and remote attestation.

Runtime Every SGX enclave features a fixed set of entry points. The host application must utilize a special SGX instruction, *EENTER*, to transfer control to the enclave. The other way around, the enclave must also utilize the special *EEXIT* instruction to return control to the host application. The entry points are defined in the so-called Thread Control Structure (TCS). However, when considering enclaves developed with the Intel SGX SDK, most enclaves only define a single entry point that is provided by the Intel SGX SDK. We defer a discussion of the particularities of the Intel SGX SDK in Section 3.1.3.

The enclave can also be exited using the asynchronous enclave exit (AEX) mechanism. This mechanism is triggered if a hardware exception occurs or an interrupt is triggered. The AEX mechanism will save the current enclave state (i.e., registers) into dedicated enclave memory before exiting the enclave mode of the CPU Afterward, and similar to normal user-space software, the usual exception-handling mechanism of the operating system is performed. However, in contrast to normal user space, the enclave's execution must be resumed with the *ERESUME* instruction.

Teardown Finally, after the enclave finishes computation, the OS can utilize the dedicated *EREMOVE* instruction to deallocate the enclave-associated pages and also delete the corresponding control structures. The corresponding pages can be reused for a newly loaded enclave only after the enclave is destroyed.

3.1.2 Threat Model and Attacks

The threat model of SGX features a very strong attacker: the whole software stack running on the CPU is considered untrusted. This means that throughout the SGX threat model, we must assume that the attacker has all capabilities of the kernel and hypervisor. At the same time, the enclave features no elevated privileges across

the system. More specifically, SGX enclaves are associated with a normal user-space process. However, they lack the capability of interacting with the outside, i.e., enclaves cannot perform system or hyper calls.

Furthermore, SGX also considers large parts of a system's hardware as untrusted. More specifically, the CPU die is considered trusted, but everything connected to the CPU package is considered untrusted. For example, the CPU caches are considered trusted, while the main DRAM memory is considered untrusted. As a consequence, the SGX technology enforces *confidentiality* of enclave pages by encrypting data that is written to main memory using an encryption key specific to the CPU. However, to speed up execution, the internal caches of the CPU are not encrypted, and the memory encryption engine is applied only to data written outside of the CPU package, i.e., main memory.

However, data confidentiality is insufficient to secure SGX enclaves. Additionally, SGX features several measures to ensure *integrity* for enclave pages. First, SGX uses access control checks to prevent any writes to the EPC. Whenever the CPU is not in enclave mode, any access to the EPC results in a hard CPU halt. Second, SGX also features cryptographic protection of memory integrity using key-dependent hashing and Merkle trees over enclave pages. This cryptographic protection also protects enclave memory against physical tampering. Notably, the official SGX threat model does not consider hardware attacks as in-scope. In that regard, cryptographic memory protection is not strictly necessary to comply with the official threat model of SGX. Interestingly, Intel introduced a variant of SGX called *scalable* SGX that lifts a hard limit of 128 Mbyte of EPC memory. However, this comes at the cost of disabling the cryptographic memory integrity protection [Sim].

Enclaves are as privileged as the host process in whose address space they are loaded. This is important to allow the operating system to defend against malicious enclaves. Since the operating system has limited introspection capabilities into SGX enclaves, it is important for the operating system to have the capability to isolate an enclave. However, the enclave itself does not have the capability to interact with the operating system because enclaves are not allowed to issue system calls. Instead, enclaves must rely on their associated host application to forward any data to the relevant system calls. The SGX technology could also be abused by malware authors to launch attacks [CD16; Sch+17]. However, due to the decision to associate enclaves with normal user space processes, the operating system can apply standard process isolation and sandboxing techniques to prevent the enclave from damaging the host system.

3.1.3 Enclave Programming Model

With the launch of the SGX technology, Intel also provided the Intel SGX SDK as the first development kit for SGX enclaves. The Intel SGX SDK has been established as the de-facto standard and is often used to provide more high-level frameworks to developers [Asy; TPV17].

The Intel SGX SDK features infrastructure to develop enclave code. The programming model of the Intel SGX SDK exposes an enclave as a *secure library*. Similar to other dynamic libraries, an enclave is loaded by a normal user space process and

exposes a set of API endpoints (i.e., library functions) that can be called by the regular untrusted user space process. In fact, the Intel SGX SDK automatically generates code that makes the transition from the host application to the SGX enclave as easy as a library call. Such a transition is dubbed *ECALL* by the Intel SGX SDK. However, the Intel SGX SDK also supports calls in the other direction, i.e., calls from the enclave to the host application. Such calls are typically used by the enclave to request input/output (I/O) operations. The Intel SGX SDK refers to this type of call as an *OCALL*.

To facilitate this programming model, the Intel SGX SDK adds standard support code and generated code to both the host application and the enclave. Conceptually, the support code is split into the trusted runtime (TRTS) in the enclave and the untrusted runtime (URTS) in the host application. To generate the support code, the build system relies on an interface specification, which is written by the enclave developer in the enclave definition language (EDL) language. The enclave developer specifies the public interface of the enclave code. The build system automatically generates wrappers around these interface functions for the URTS. The host application can call these wrapper functions like normal C functions. The wrappers of the URTS then transparently pack the arguments according to the calling convention expected by the TRTS. The URTS enters the enclave using the SGX specific instructions. Now the TRTS takes over and performs unpacking and validation of the parameters of the ECALL. In fact, the TRTS only supports a single entry point and multiplexes all logical ECALLs. After unpacking and validation, the TRTS passes control to the ECALL handler that is specified by the enclave developer. When the ECALL handler finishes execution, control is passed back to the TRTS, which writes the returned data back to untrusted memory, before exiting the enclave to continue execution in the URTS. To perform an OCALL the flow is quite similar, except that now the TRTS begins by packing the OCALL parameters to untrusted host application memory. Furthermore, the TRTS unpacks and validates the returned data of an OCALL. Technically, a return from an OCALL can be thought of as a special ECALL and reuses much of the same mechanisms.

The EDL Interface Specifications The Intel SGX SDK uses the EDL, a custom specification language, to define the ECALL and OCALL interface of an enclave. More specifically, the EDL file is similar to a header file in the C language but with syntax to represent SGX specific terminology. For example, it uses the *trusted* and *untrusted* keywords to define ECALLs and OCALLs. Each ECALL or OCALL is defined by the function prototype akin to a C function declaration. However, in addition to the usual C parameter type definition, EDL features additional annotations. For example, Figure 3.1 shows various ECALL and OCALL definitions.

Based on the provided EDL files, the Intel SGX SDK generates the wrapper code to connect the ECALL stubs in the host application with the ECALL handler in the enclave. When the host application calls an ECALL stub, the auto-generated code packs the parameters of ECALLs into auto-generated data structures in the host application memory. Control is then transferred to the TRTS and the auto-generated code on the enclave side, which unpacks and validates the parameters from host

```
enclave {
  struct foo_t {
    uint32_t part1;
    uint64_t part2;
  };

  trusted { // ECALLs
    public void ecall_size1( // explicit size
        [in, size=100] void* ptr);
    public void ecall_size2( // variable size in len
        [in, size=len] void* ptr, size_t len);
    public void ecall_user( // dangerous user_check
        [user_check] void* ptr);
    // passing a struct by value
    public int ecall_strct(struct foo_t foo);
  };

  untrusted { // OCALLs
    void untrusted_print([in, string] const char *msg);
  };
};
```

Figure 3.1: Example for the EDL syntax.

application memory and passes control to the ECALL handler. The enclave developer can specify various types of parameters in EDL, including pointer parameters and structures. Especially for pointer parameters, the Intel SGX SDK must be able to determine the size of the underlying data to copy it to enclave memory before passing it to the ECALL handler. Thus, every pointer parameter is also annotated with how the size is to be determined. Figure 3.1 shows several of these size annotations, e.g., constant size, C-style string length, and a second length parameter. Currently, the Intel SGX SDK supports standard C data types, such as basic integers types, composed data types (i.e., *struct*) without nested pointers, enumerations (i.e., *enum*), null-terminated strings, and pointers to arrays of fixed length.

Figure 3.1 shows various features of the EDL language. For example, here, a `void*` is passed to the enclave. The Intel SGX SDK requires a length annotation to determine the underlying buffer size: `[in, size=100]`. This annotation specifies that the underlying buffer is 100 byte. As such, the Intel SGX SDK will generate code that copies this 100 byte buffer from the untrusted host application to enclave memory. The ECALL handler is then passed a `void*`, that points to the corresponding buffer in enclave memory.

However, there are many particularities that have to be considered when writing EDL interface specifications. The most particular feature is the `[user_check]` features, which completely disables any validation by the Intel SGX SDK. This annotation is used to pass in raw pointer parameters to the ECALL handler. This leaves the burden of validating this data to the enclave developer. In Chapter 4, we discuss several memory corruption vulnerabilities that are due to missing validation of such raw pointer ECALL parameters.

Furthermore, the Intel SGX SDK does not perform deep copies of compound data types. In fact, any *struct* is treated as a buffer of a fixed size. As a consequence, the Intel SGX SDK also does not support passing container types, such as the containers available in the C++ standard library (i.e., *vector* or *map*).

3.1.4 SGX Vulnerabilities

The security of SGX has been extensively studied. We have seen two orthogonal research directions: (1) attacks against the SGX technology, and (2) attacks against the software inside of SGX enclaves.

Numerous works study the security of the SGX technology itself. SGX is implemented as an extension to the regular x86 ISA. In recent years a significant amount of work has studied the security of the ISA implementations, i.e., the micro-architecture. As such, there exist many attack vectors that exploit micro-architectural features, such as caches, branch predictors, port contention, and so on [Can+19; Gru20]. Most micro-architectural attacks require the attacker and victim to execute on the same machine, which makes this type of attack typically hard to execute. However, the threat model of SGX offers the perfect environment for micro-architectural attacks: by design, the attacker, the untrusted host, executes on the same CPU as the victim, the enclave. Furthermore, in SGX, the attacker has an incredible amount of control over the enclave, which is executed as part of an unprivileged user space process.

We have seen multiple attacks that exploit micro-architectural features to leak data from SGX enclaves, such as cache attacks [Bra+17] or transient execution attacks [Che+19a; Van+18]. At least two of the attacks also targeted special enclaves developed by Intel. As part of the remote attestation, a special SGX *quoting enclave* issued and signed by Intel is used to obtain and sign a measurement of a local enclave. Attacking these special enclaves with microarchitectural side-channel attacks allowed researchers to dump the signing keys used by these enclaves [Sch+20a; Van+18], which allows the attacker to subvert the whole remote attestation protocol. Intel has continuously deployed new defenses against various micro-architectural attacks against their CPUs and SGX in particular. Given the current state of attacks, it is questionable whether the SGX technology can hold up to its threat model in practice.

A second line of research has assessed the security of the enclave code itself. The code running inside a SGX enclave also faces a new threat model due to the limited control over its own execution. Enclaves are prone to so-called controlled-channel attacks [XCP15], a deterministic form of side-channel attack. For example, a SGX enclave must assume that it can be interrupted by the host OS at every single instruction [VPS17]. Furthermore, the enclave must assume that the host OS knows exactly which memory pages are kept in main memory and which have been moved to disk. This allows the attacker to deduce which parts of the code have been executed and which of the data pages are accessed. Even worse, combined with side-channel attacks, a controlled-channel attack can reconstruct the precise control flow of an SGX enclave [Mog+20]. In turn, this means that secret-dependent branching can lead to an enclave directly leaking the secret to the attacker.

However, the SGX environment poses further challenges for the enclave software: the enclave needs to securely transition between the untrusted and trusted execution

domain. While the hardware enforces isolation, the security of the transition must be secured in software. The transition between the two execution domains is defined by two parts: the application binary interface (ABI) and the API. The ABI defines the low-level calling conventions used to pass data via registers or memory between the two domains. One abstraction level higher, the API defines a logical interface that usually resembles a function call. Both have been targeted by prior work [Ald+20; Van+19]. It was shown that lacking sanitization of status flag registers or the status of the floating point unit could be abused to leak data from an enclave. Furthermore, several enclave runtime environments lacked proper automatic sanitization of data passed to the enclave or returned by calls to the untrusted world.

The first programming environment for SGX was the Intel SGX SDK, which targets the C/C++ language. While this offers the greatest flexibility and compatibility, it also means that memory safety is not automatically enforced in SGX enclaves. As such, prior work has analyzed exploit techniques targeting memory corruption in the context of SGX enclaves [Bio+18; Lee+17]. Similar to classical x86 software, SGX enclaves are also prone to code-reuse attacks such as return-oriented programming (ROP). However, there was no prior investigation into the prevalence of memory corruption errors. In Chapter 4, we will discuss our results when analyzing several enclaves and the new vulnerability pattern that we found.

3.2 The Ethereum Execution Environment

A blockchain commonly refers to a distributed ledger, which records transactions and is replicated across all nodes participating in the network. Traditionally, transactions involve transferral of the currency associated with the blockchain platform but can also contain other arbitrary data. Typically, a transaction is defined to contain a *sender*, a *receiver*, the *amount* to be transferred, optionally attached data, and a digital signature over the prior transaction data created by the sender. Multiple transactions are usually collected into a *block*. Multiple blocks are then chained using cryptographic hash functions. Using a consensus mechanism, such as the Proof-of-Work mechanism introduced by Bitcoin [Nak08], the network converges to the longest block sequence as the active state of the ledger. This process is called *mining* and usually involves rewarding the miner in terms of new cryptocurrency. As the first major blockchain platform, Bitcoin has already added support for attaching arbitrary data to transactions. Soon this was extended to attach program code to transactions: so-called *smart contracts*. Smart contracts in Bitcoin are written in stack-based transaction scripts. These scripts were initially intended only for basic functionality: defining the eligible owner(s) and also transferring this ownership. However, combined with Bitcoin's time-lock feature, more complicated protocols can be implemented via the Bitcoin scripts. While the restrictions of the bitcoin scripting language only allowed very specific applications, the idea of executing code on top of a blockchain prevailed.

Ethereum [Woo19], a second-generation blockchain system, introduced a Turing-complete programming environment for smart contracts alongside their cryptocurrency *Ether*. Furthermore, smart contracts are designed to be first-class citizens in the Ethereum blockchain: similar to normal user accounts, a smart contract is associated

with an address, can manage its own currency balance, and can issue transactions to other accounts (including smart contracts).

While, for the most part, smart contracts are first-class citizens in the Ethereum blockchain, there are several differences between a smart contract and an externally-owned account (EOA). Smart contracts are strictly reactive: any smart contract execution is triggered by an initial transaction issued by an EOA, e.g., a normal user. Although functionally equivalent, transactions created by a smart contracts are treated a bit differently in the Ethereum protocol. They are usually called *internal transactions* as they are not explicitly recorded in the blockchain but can only be observed as sub-transactions when executing transactions issued by an EOA. The EOA that issues a transaction also pays the gas fees for all internal transactions that might be executed due to smart contracts reacting to the top-level transaction. Additionally, smart contracts can store state directly on the blockchain, which is impossible for EOA accounts.

Like traditional contracts, smart contracts are also considered immutable once they have been created. All parties that interact with a smart contract commit implicitly to the smart contract's code. Similar to traditional contracts, changes to smart contracts require creating a new contract. We discuss the problems of smart contract immutability and upgrade strategies in detail in Chapter 7.

Due to the distributed nature of a blockchain platform, several restrictions have to be imposed on the execution of Ethereum smart contracts:

1. Smart contract execution must be completely deterministic.

2. Smart contract execution must be bounded in space and time.

We will now discuss these two restrictions and how they are implemented in Ethereum.

Deterministic Execution In Ethereum, smart contracts must be completely deterministic. The current blockchain state is computed by applying all recorded transactions in the same order to the initial blockchain state, typically called the *genesis* state. Applying a transaction that involves a smart contract also implies that the smart contract code is executed to obtain the next blockchain state. Since any full node that joins the Ethereum network must compute the current blockchain state based on the genesis state, the execution of a smart contract must be completely deterministic. If, at any point in time, the execution of a transaction targeting a smart contract leads to a different outcome, the whole protocol breaks down as this would violate the cryptographic mechanism chaining the blocks based on their cryptographic hashes. As a consequence, the Ethereum execution environment is carefully designed to only allow deterministic behavior. Any inputs and outputs are well-defined and are part of the blockchain state. Furthermore, common features in normal programming environments are impossible: there can be no source for random number generation and no machine-specific behavior. For example, IEEE floating point arithmetic is not supported, as floating point arithmetic can lead to different results depending on the implementation in the CPU as small rounding errors are allowed. Instead, Ethereum smart contracts use deterministic fixed-point arithmetic.

Bounding Execution with Gas Since the execution of a smart contract must be replicated within the whole network, it is essential to limit the execution of smart contracts in time and space. Otherwise, a simple endless loop of a single smart contract would immediately take down the whole system. Similarly, since the blockchain is an append-only data structure, care must be taken not to unnecessarily increase the physical storage space required to store the blockchain. As such, the size of the smart contract's state must also be bounded. Ethereum tackles both problems using the *gas* mechanism and the derived transaction costs. Whenever a transaction is committed to the blockchain through mining, the EOA that originally sent the transaction must pay for the cost of executing the transaction using their available cryptocurrency. The cost of a transaction is computed using the *gas* mechanism. Each operation of a smart contract has an associated *gas* consumption. The EOA origin of a transaction then specifies an upper bound for total gas consumption and a gas price in Ether per gas unit. The balance of the transaction sender is then reduced by the consumed gas times the gas price and transferred to the *miner* of the block that contains the transaction. As such, the increasing cost of the used gas naturally limits the resources used by smart contracts. However, miners also impose a block gas limit, which limits the number and execution complexity of transactions per block.

3.2.1 Ethereum Virtual Machine

Ethereum specifies a custom virtual machine and bytecode format: the EVM. The EVM is a deterministic virtual machine without any source of randomness or behavior specific to the CPU. Furthermore, the EVM was designed with simplicity in mind. For the consensus layer it is essential that all nodes, which might run different implementations, can replicate the exact same EVM executions. By designing the EVM ISA as simple as possible, it is ensured that implementations are more likely to be correct.

The EVM is a stack-based architecture: every instruction pops arguments from the stack and pushes the result back on top of the stack. The default word size for the EVM is 256 bit. As such, all stack operands utilize this bit width. Instructions are encoded as one-byte opcodes, except for the `PUSH` family of opcodes, which are used to push constants. Figure 3.2 shows an example for parts of an execution trace of a EVM smart contracts. It contains several notable instructions that show certain particularities of the EVM. First, we have the `JUMPDEST` pseudo-instruction. While it is technically a no-op instruction, it is used in the EVM to mark valid jump destinations. The EVM enforces a form of very coarse-grained control-flow integrity (CFI) [Aba+05], which allows jumps to target only to program locations that have been previously marked with the `JUMPDEST` pseudo-instruction. While not generally considered a security mechanism, this property also prevents code-reuse attack techniques such as ROP [Sha07; Sze+13]. Second, we have the `PUSH2` instruction, which places a 2-byte constant on top of the stack. This instruction is encoded as 3 byte in the bytecode format of the EVM. Interestingly, there is another restriction in the EVM when it comes to jump targets. The EVM forbids jumps into the constants that are encoded as part of `PUSH` instructions.

In contrast to many modern computer architectures, the EVM resembles a Harvard architecture that separates code and data into isolated address spaces. Once the

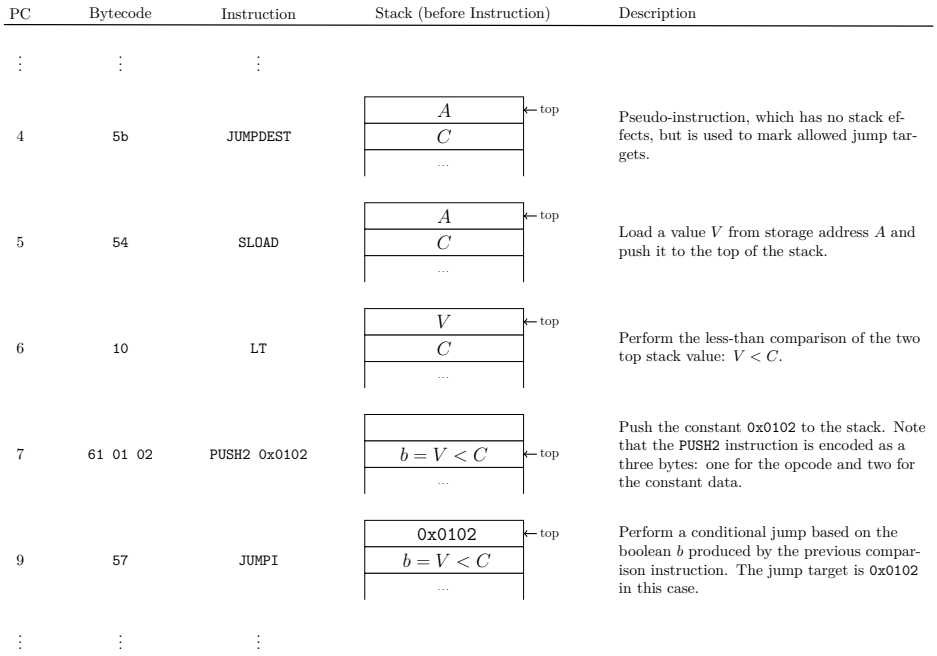


Figure 3.2: Example for a part of an EVM execution trace, including the program counter (PC), the bytecode, the instruction mnemonic, a depiction of the stack, and a description of the effects of the instruction.

contract is created, its code is immutable and cannot be written anymore. The only exception is the final destruction of a contract, which removes the code of the contract. In fact, the EVM not only separates code and data, it uses multiple isolated address spaces for different types of memory for data. The EVM separates its memory into the following memory sections:

- The stack.
- *Memory*, an area for dynamic memory allocation during execution.
- *Storage*, for persistent data that is part of the blockchain state.
- The *call data*, which contains the input attached to a transaction.
- The *return data*, which contains the data returned by the last call.
- The code, which is read-only but can be accessed to retrieve larger constants such as fixed strings or constant binary data.

Most of the address spaces are byte-addressable in current versions of Ethereum. Two notable exceptions are the stack and the storage. The stack is not directly accessible except but can only be manipulated with instructions pushing and popping and with the `DUP` and `SWAP` opcodes that duplicate and swap 256 bit stack slots relative to the top of the stack. Addresses in the storage address space point to full 256 bit

words, i.e., it can be thought of as a map data structure that maps 256 bit keys to 256 bit values. In fact, hash maps in the storage area can be easily implemented in the EVM with the help of the `SHA3` opcode and are commonly used in smart contracts.

Lifecycle of a smart contracts Smart contracts are created by an EOA issuing a transaction with an empty *receiver*. This prompts the Ethereum miners to execute a creation transaction for the smart contract. First, the miner computes the new address of the smart contract and then executes the constructor bytecode. In Ethereum, the input attached to the creation transaction is treated as the *constructor* code. This bytecode is executed once in the EVM environment to initialize a smart contract. Notably, the constructor execution is performed in the exact same environment as the regular smart contract execution. As such, the constructor is allowed to issue further transactions, i.e., to call other smart contracts. However, from the point of view of all other smart contracts, a smart contract that is currently being constructed has no code attached. Incidentally, this makes it impossible to reliably distinguish smart contracts from EOA in nested transactions, which has led to bugs in the past.

Once a smart contract has been created, it is essentially immutable: the code cannot change anymore. Now anyone participating in the Ethereum network can interact with the smart contract by issuing transactions, whose *receiver* field is the address of the smart contract. As such, Ethereum smart contracts are always available to anyone by design, including attackers. By default, there is no way to temporarily take down a smart contract while a patch for a vulnerability is in development. This makes patching highly challenging, which we will discuss in more detail in Chapter 7.

Finally, the life cycle of a smart contract ends when it is destroyed. While most smart contracts do not use this feature, it is beneficial to deprecate and destroy old unused smart contracts because this reduces the size of the current blockchain state at the latest block. A contract must voluntarily destroy itself using a special `SELFDESTRUCT` instruction. This instruction transfers all remaining Ether associated with the smart contract to a given address and then deletes the code and data associated with the smart contract.

3.2.2 Programming Paradigms in Ethereum

Ethereum smart contracts are primarily programmed in *Solidity* [Tea18], a programming language that features a C-like syntax and the contract-oriented programming paradigm. Contracts are treated similarly to objects in the common object-oriented programming paradigm. Contracts have associated methods and variable members and can inherit from other contracts. While Solidity is the prevalent smart contract language, other languages like Vyper [Vyp] or Fe [Fou22] have been developed.

However, Solidity has been criticized for its non-intuitive syntax that easily allows introducing security bugs into the smart contracts [Dai16b; Luu+16]. Major hacking incidents like the DAO attack [Dai16a] or the Parity Wallet attack [Bre+17; Tec17b] can be traced back to issues with the Solidity language. Even though the volume of cryptocurrency managed by Solidity smart contracts is tremendous [Eth22], Solidity has not yet reached a stable release version of *1.0* [Tea22b]. As such, the language is still undergoing significant changes and has fixed many of the previous criticisms.

Solidity, and most other smart contract programming languages, expose a set of *methods*, which operate on the provided input parameters and the contract's state. However, on the EVM level, there is no notion of functions or methods: A smart contract always starts executing at code address zero. Similarly, the input data attached to the transaction does not distinguish between parameters. The EVM treats the transaction input data simply as a sequence of bytes. However, the input to most smart contracts must follow a certain structure, which is decoded by the smart contract itself. Solidity has defined a custom ABI for source languages targeting the EVM [Tea22a]. This ABI specifies how the transaction input must be formatted when a certain function should be called. More specifically, the first four bytes of the EVM input contain a function selector. In Solidity, this function selector is computed as a hash over the type signature of the method. The Solidity compiler emits a dispatcher at the start of the smart contract, which jumps to the right function based on the function selector in the input. A programmer can also specify a fallback function that is called when no, or no known, function selector is provided.

One important feature of Solidity smart contracts is the possibility to interact with other smart contracts. In fact, smart contracts have multiple different ways to interact with each other. On the EVM level, there are multiple instructions that are used to issue calls between smart contracts. Most importantly, the normal `CALL` instruction issues a new internal transaction that is used to a) transfer Ether to EOA and other smart contracts, and b) call public methods of other smart contracts. However, there are also multiple other types of calls. The `DELEGATECALL` instruction[1] is used to call other smart contracts as libraries, i.e., the library smart contract is executed with the state of caller smart contract. The `STATICCALL` is like the normal call instruction but prevents the callee from performing state updates. This is useful when the intention is only to retrieve data from the other smart contract, e.g., a *getter* method. Furthermore, smart contracts can utilize the `CREATE` family of instructions to deploy new smart contracts.

Somewhat confusingly, on the EVM level, all the call instructions are concerned with performing external calls, i.e., calls targeting other smart contracts. Even though high-level languages like Solidity fully support this, there is no instruction to perform internal function calls within a smart contract's code like in traditional ISAs. While there are proposals to add such instructions [Bro22], they are not yet in widespread use at the time of writing. As a result, many compilers emulate internal calls by pushing a return address to the stack and returning by using the normal `JUMP` instruction. Note that this causes significant problems when statically analyzing EVM bytecode. Since a normal jump cannot be distinguished from a return, it is highly challenging to build an accurate control-flow graph (CFG) for even moderately complex EVM bytecode. Several research projects have attempted to tackle this issue using various static analysis methods [Alb+18; Bre+18; Con+21; Gre+19].

[1] and the similar but deprecated `CALLCODE`

```
function batchTransfer(address[] _receivers, uint256 _value)
                        public whenNotPaused returns (bool)
{
  uint cnt = _receivers.length;
// OVERFLOW: 2 * ((INT_MAX / 2) + 1) == 0
  uint256 amount = uint256(cnt) * _value;
  require(cnt > 0 && cnt <= 20);
// BYPASSED CHECK: balances[msg.sender] >= 0
  require(_value > 0 && balances[msg.sender] >= amount);
  /* .... */
// RESULT: Transfer of ((INT_MAX / 2) + 1) tokens
```

Figure 3.3: Integer overflow bug reported by PeckShield [Peca].

3.2.3 Smart Contract Vulnerabilities

Over the history of the Ethereum blockchain platform, there have been various incidents due to vulnerabilities in smart contracts [ABC17; Luu+16; SWC]. The attacks range from denial-of-service (e.g., the King of the Ether Throne bug [KotET]), unavailable Ether (e.g., the Parity multisig wallet [Tec17a]), to attackers being able to steal Ether (e.g., the DAO attack [Jen16]). There are numerous potential pitfalls and various types of bugs in Ethereum and Solidity. In the remainder of this section, we will discuss the most important bug classes, which are tackled in the following chapters of this dissertation.

Integer Bugs Due to performance reasons and engineering constraints, typical CPU ISAs utilize fixed bit-width integers. In this respect, the EVM architecture is no different, even though the EVM uses an unusually large bit-width of 256 bit. Essentially, this means that the mathematical integer is only approximated by computers. This can lead to bugs at the boundaries of the capabilities of fixed bit-width integers. When considering smart contracts, which generally need to track cryptocurrency or token balances, integer bugs can become quite critical vulnerabilities. For example, there have been multiple reports on integer-related bugs in *token* contracts [FSS18; Peca].

In the Ethereum ecosystem, token contracts implement specialized sub-currencies. Users can utilize exchanges to convert Ether, the native currency, into tokens. Once the user possesses tokens, they can transfer them by calling the respective method of the token smart contract. Figure 3.3 shows an example of an integer overflow vulnerability in a token contract. The token features a *batchTransfer* function, which allows transferring a certain number of tokens to multiple other accounts at once. However, there is an integer overflow bug, which the attacker can exploit to overcome a balance check. More specifically, the attacker can choose the parameters to the vulnerable function, such that the total amount of needed tokens—stored as *amount* in Figure 3.3—becomes 0. As a result, the attacker can trick the contract into transferring a large number of tokens without owning a single token.

Besides, integer overflows Ferreira-Torres et al. [FSS18] identify three major types of integer bugs that are also applicable to smart contracts:

Arithmetic Bugs include integer over- and underflows, i.e., the fixed bit-width integer causes a wrap-around because the result of an arithmetic computation would exceed the capability of the fixed-width integer type.

Truncation Bugs happen when converting between a larger integer type to a smaller one, discarding critical information in the process. For example, if a 256 bit integer larger than 255 is converted to an 8 bit integer, the upper bits must be discarded.

Signedness Bugs happen if an unsigned integer type is converted to a signed integer type or the other way around. On the EVM level, signed integers are represented using the common two's complement approach. As a result, converting a small signed integer will result in a very large unsigned integer.

Access Control Bugs To maintain a smart contract, many developers introduce the *Owned* contract pattern. Here the contract maintains an address of a special *owner* account. Access to specially privileged functions is then restricted to originate from the owner. However, there is no architectural support or support from the language. As such, any access control must be implemented manually by the developer. Especially in early smart contracts, this often caused bugs if the access control checks were only partially applied or became ineffective due to unrelated changes to the smart contracts. This type of bug belongs to one of the most well-studied bug classes in the smart contract literature [FAH20; KR18a; Luu+16; Nik+18; SHO21]. These studies identified many vulnerable contracts that were deployed to the blockchain.

This type of vulnerability received special attention after the infamous *parity wallet* incident, where many funds were vulnerable and finally frozen due to bugs in the code and issues during deployment. Figure 3.4 shows an example of an access control bug. Here the *Wallet* smart contract utilizes a library smart contract that introduces a set of *owners* with special permissions. The library contract exposes the *initWallet* function to initialize the *owners* array, which allows lazy initialization of the contract. Normally, this function is supposed to be called once in the constructor of the contract that uses the library. However, this initialization function is not properly protected by access control checks. As such, anybody can call the *initWallet* function even after the first initialization. This allows any participant of the Ethereum network to overtake the *Wallet* contract by calling the *initWallet* with a custom set of *owners*.

Reentrancy Vulnerabilities Reentrancy vulnerabilities have been the root cause for several high-profile attacks on the Ethereum blockchain. The most infamous *TheDAO* incident [Dai16a], the *SpankChain* incident [Spa18], and the more recent *Uniswap* and *Lendf.Me* incidents [Tor+21b]. In Ethereum, a reentrancy bug is a deterministic form of a common concurrency problem where a contract calls another contract which calls back into the calling contract. However, if the contract is not safe to be re-entered, then reentrant calls operate on potentially invalid internal state of the smart contract, i.e., state is only partially updated by the previous invocation. Such inconsistent state can then be exploited by an attacker to bypass critical checks.

```
contract WalletLibrary {
  address[16] owners;
  mapping(bytes => uint256) approvals;

  function confirm(bytes32 _op) internal returns(bool) {
    /* logic for confirmation */
  }
  function initWallet(address[] memory _owners) public {
    /* initialize the Wallet functionality */
  }
  function pay(address payable to, uint amount) public {
    if (confirm(keccak256(msg.data))) {
      to.transfer(amount);
    }
  }
  // ...
}
contract Wallet {
  // the Wallet contract utilizes the WalletLibrary contract
  // deployed at a fixed address:
  address walletLibrary = address(0x1234ABCDFEFEFE);
  constructor(address[] memory _owners) {
    // initialize the Wallet functionality
    bytes memory data = abi.encodeWithSignature("initWallet(address[])",
    ↪  _owners);
    walletLibrary.delegatecall(data);
    // ...
  }
  // forward all unknown calls to the WalletLibrary
  fallback() external payable {
    walletLibrary.delegatecall(msg.data);
  }
}
```

Figure 3.4: A simplified variant of an access control issue similar to the bug of the *parity multisig wallet*.

However, reentrancy is also often occurring during normal contract execution. For example, it is part of common and officially supported programming patterns for Ethereum smart contracts [SolWd]. The common withdrawal code pattern [SolWd] is shown in Figure 3.5 Here contract A withdraws 100 wei from contract B. In Figure 3.5, contract A calls the public *withdraw* function of contract B. During this call B then invokes the built-in *call* method on the *sender* (`msg.sender` in Solidity). This then transfers the specified amount to A (i.e., *msg.sender* is representing the calling contract A). In Ethereum, Ether is transferred by means of a call, e.g., contract B must call back (*re-enter*) into contract A's fallback or receive function to send the funds.

There are several ways to call other smart contracts in Solidity. First, there are the low-level call mechanisms, like *send*, *call*, and *transfer*, which are part of the *address* type in Solidity. Generally, the send and transfer functions are considered reentrancy safe, as they only provide a limited amount of gas to the called contract, effectively

```
contract A {
  B b;
  function f() public {
    b.withdraw(100);
  }
  fallback() external payable { }
  // ..
}

contract B {
  function withdraw(uint amount) public {
    msg.sender.call{value: amount}("");
  }
}
```

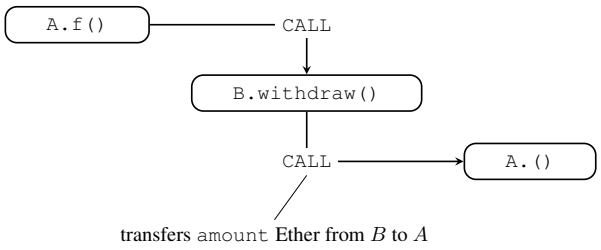


Figure 3.5: Common withdrawal pattern in Solidity: the upper part shows the sample Solidity code, whereas the lower part shows the call chain. In this example contract A withdraws 100 wei from contract B.

preventing the callee contract from performing further calls. However, when an address is cast to a smart contract (or its interface), then the smart contract is usually called via the regular *call* mechanism that is not reentrancy safe. The exception is when a method is called that is annotated as *pure* or *view* in Solidity. These types of methods are called using the `STATICCALL` instruction, which prevents state updates and further external calls. As such, these calls can be considered reentrancy safe.

A malicious reentrancy occurs when a contract is reentered *unexpectedly* and the contract operates on *inconsistent* internal state, i.e., the contract accesses persistent data in the storage area that is not consistent with a concurrent execution of the smart contract. More specifically, we define a reentrancy bug to occur when the following conditions are met:

1. The contract is called at call depth C_1 and is reentered at call depth C_2, where $C_1 < C_2$.

2. The contract loads a value V from storage address A_S.

3. The contract writes a value V' with $V' \neq V$ to storage address A_S after the reentrant execution at C_2 returned.

```
function withdraw(uint amount) public {
 ①  if (credit[msg.sender] >= amount) {
 ②    msg.sender.call.value(amount)();
 ③    credit[msg.sender] -= amount;
  }
}
```

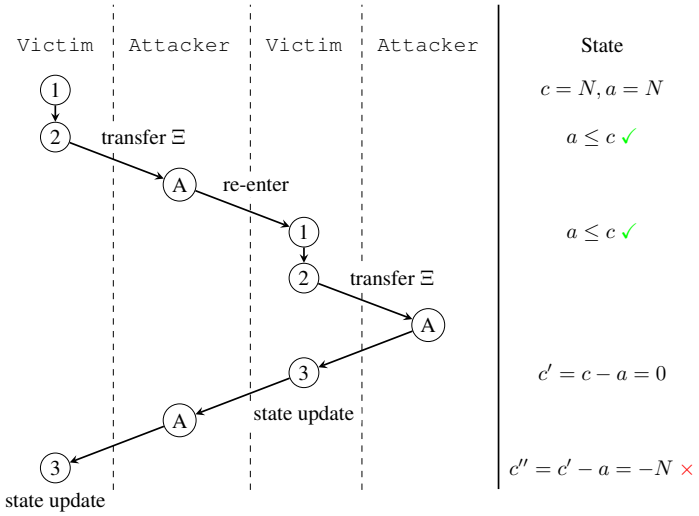


Figure 3.6: Sample contract vulnerable to re-entrancy attacks: the upper parts shows the Solidity code, whereas the lower part shows the call sequence between the vulnerable contract `Victim` and the attacker contract, and the state of the variable a (amount) and c (`credit[msg.sender]`). The amount a has not been updated for the second invocation of `Victim` thereby allowing a malicious re-entrancy.

4. There is data-dependency on V from a control-flow instruction, a storage write, a `RETURN` instruction, or one of the parameters to one of the call instructions, and one of these instructions is executed in the reentrant call at depth C_2.

This implies that the reentered victim contract operated based on an inconsistent state value, and thus the reentrancy was not expected by the contract developer.

From this definition, one can deduce a rather simple policy to prevent reentrancy issues: *disallowing state updateds after calls*. Here, it is forbidden to write to storage addresses after an external call happens, i.e., after potential reentrant executions. By design, this ensures that a reentrant execution always operates on the latest state of the contract. While this policy is implemented by several static and symbolic analysis tools, such as Slither [FGG19], Securify [Tsa+18] or Mythril [Conc], this policy is also overly restrictive and suffer from a significant number of false alarms (see Section 6.5). In many cases, it is not possible to remove the state update after the call. For example, whenever a state update depends on the return value of an external call, this policy cannot be enforced.

Figure 3.6 shows a simplified version of a contract (inspired by Atzei et al. [ABC17]), called *Victim*, which suffers from a reentrancy vulnerability. *Victim* keeps track of an amount (a) and features the *withdraw* function allowing other contracts to withdraw Ether (c). The *withdraw* function must perform three steps: ① check whether the calling contract is allowed to withdraw the requested amount of Ether, e.g., checking whether $a \leq c$, ② send the amount of Ether to the calling contract and ③ update the internal state to reflect the new amount, e.g., $c - a$. Note that step ② is performed before the state is updated in ③ . Hence, a malicious contract can re-enter the contract and call `withdraw` based on the same conditions and amounts as for the first invocation. As such, an attacker can repeatedly reenter into *Victim* to transfer large amounts of Ether until the *Victim* is drained of Ether. A secure version of our simple example requires swapping lines 3 and 4 to ensure that the second invocation of *Victim* operates on a consistent state with updated amounts.

3.2.4 Identifying Basic Blocks in EVM Bytecode

A typical Ethereum client must be able to quickly load and execute many different smart contracts. As such, the overhead of loading a smart contract must be small. At the same time, the execution speed of the smart contract should be minimal. As such, the EVM bytecode exhibits several properties that make it amendable for quick analysis. For example, while it is notoriously difficult to properly build a CFG from EVM bytecode [Alb+18], it is possible to identify basic block boundaries using a single linear pass. This eases building faster interpreters or helps to reduce the latency of just-in-time compilers. In Algorithm 1, we describe the algorithm to identify all basic blocks, all legitimate jump targets, and all data constants within EVM bytecode.

The EVM bytecode format is quite simple, as every opcode is simply a single-byte identifier. Since everything is stack-based, there is no encoding of register identifiers or similar metadata into the opcode itself. However, there is one exception: EVM implementations prevent jumps into the data constants that are embedded into `PUSH` instructions. Here the EVM enforces a slight separation of code and data in the code address space. The constant operands of the push instructions follow directly after the byte of the push instruction opcode. Such a constant operand could include the byte for the `JUMPDEST` instruction. Then, the constant would be a legitimate jump target. As a consequence, a new unintended instruction sequence would occur. Since such unintended instruction sequences can have a negative impact on security, as on other native CPU architectures [Sha07], the EVM attempts to avoid them. To avoid such unintended instruction sequences, EVM implementations must perform a linear sweep as shown in Algorithm 1 over the code section to find all push instructions. The constants that are part of those push instructions are then marked as data. They become invalid jump targets, even if they contain a byte equivalent to the `JUMPDEST` instruction.

However, push constants are not the only data inside the code address space. Many smart contracts also embed constant strings or the constructor code of other sub-contracts into their code segment. Both must be considered as data in the context of the current contract bytecode. However, there is no explicit marker for distinguishing such things as data within the code address space. Furthermore, due to performance

Algorithm 1 Identify all basic blocks in EVM bytecode.

Input: EVM Bytecode of length N
Output: BB, J, D

1: $BB \leftarrow \emptyset$ ▷ Set of basic blocks $(pc_{\text{start}}, pc_{\text{end}})$
2: $D \leftarrow \emptyset$ ▷ Data constants ranges $(pc_{\text{start}}, pc_{\text{end}})$
3: $J \leftarrow \emptyset$ ▷ Allowed jump destinations, set of pc offsets
4: $pc \leftarrow 0$ ▷ Current offset into the bytecode
5: $pc' \leftarrow pc$ ▷ Start of current basic block
6: **while** $pc < N$ **do**
7: **if** get_op(pc) = `JUMPDEST` **then**
8: $J \leftarrow J \cup \{pc\}$
9: $BB \leftarrow BB \cup \{(pc', pc - 1)\}$
10: $pc' \leftarrow pc$
11: **else if** get_op(pc) $\in$ { `JUMP`, `JUMPI` } **then**
12: $BB \leftarrow BB \cup \{(pc', pc)\}$
13: $pc' \leftarrow pc + 1$
14: **else if** get_op(pc) $\in$ {`PUSH1`, ..., `PUSH32`} **then**
15: $s \leftarrow$ get_push_size(pc)
16: **if** $pc + 1 < N$ **then**
17: $D \leftarrow D \cup \{(pc + 1, pc + 1 + s)\}$
18: $pc \leftarrow pc + s$ ▷ Skip constant data when scanning instructions
19: $pc \leftarrow pc + 1$

reasons, EVM implementations ignore control-flow information when marking data, i.e., they rely on a single linear sweep as shown in Algorithm 1. As such, the push instructions opcode byte itself can be part of some data constant, such as a string or other binary data, marking the following bytes in the bytecode as invalid jump destinations. For this reason, current smart contract compilers accumulate all data constants at addresses strictly larger than any reachable code, avoiding any conflicts between the generated code and data encoded into the code address space. This also means that any analysis that uses Algorithm 1 must be aware that the set of basic blocks will contain invalid, incomplete, and generally garbage basic blocks at the end of the contract bytecode and must handle this accordingly.

3.3 Comparison of Secure Execution Environments

In this chapter, we discuss two new execution environments for software: *SGX* in Section 3.1 and *Ethereum* in Section 3.2. Both execution environments promise a form of secure execution to the developer. We give an overview of their features in Table 3.1. While both execution environments feature quite distinct features, they do essentially offer the same programming model for software inside of the secure execution environment: a library-like API. A user, or client, of a trusted SGX enclave or Ethereum smart contract interacts with the trusted software using an API. On the logical level, this API features several function definitions along with their parameters. However, in terms of delivery, both environments are quite different. A function call in Ethereum entails creating, cryptographically signing, and finally broadcasting a transaction to the Ethereum peer-to-peer network. After the mining process is finished, the execution is done and can be inspected. In contrast, SGX enclaves are loaded as part of a local environment, a normal user space process. This user space process then utilizes the SGX hardware extensions to jump into the native enclave code.

The nature of the open and permission-less Ethereum blockchain leads to one significant drawback: there is no confidentiality. In order for the blockchain protocol to operate correctly, any smart contract code and state must be public. In contrast, SGX is designed to allow for *confidential computing*, i.e., nobody should be able to observe an enclave's internal state or data. However, code and data integrity is required to achieve confidentiality, or otherwise, an active attacker can subvert the enclave's code or inject faults to divert the enclave's execution flow.

Table 3.1: Comparison of Secure Execution Environments

	Ethereum	SGX
Code Integrity	Yes	Yes
Code Confidentiality	No	Partial
Data Integrity	Yes	Yes
Data Confidentiality	No	Yes
Trusted Third Party	No	Yes (Intel)
Enforcement	Consensus	Hardware

CHAPTER 4

SYMBOLIC EXECUTION OF SGX ENCLAVES

In this chapter, we discuss analyzing SGX enclave code using symbolic execution. To investigate the prevalence of memory corruption in SGX enclaves, we developed the symbolic executor TeeRex (Section 4.1), which is capable of identifying exploitable memory errors in SGX enclaves. We show how to apply TeeRex to real enclaves, identifying multiple vulnerabilities. Furthermore, we perform root-cause analysis on the discovered vulnerabilities and summarize common vulnerability patterns in enclave code.

The design of SGX links enclaves to a corresponding untrusted *host application* (see Section 3.1 for background on SGX). More specifically, the enclaves in the programming model of the Intel SGX SDK can be used like a shared library. The Intel SGX SDK offers a C-function-like interface allowing bidirectional communication between the host application and the enclave. Effectively, SGX introduces two privilege levels into the regular address space of a program: the privileged enclave and the unprivileged host application. When software is partitioned into privilege levels, it is essential to secure the interface between privilege levels [CS13; Hu+15]. More specifically, the interface between enclave and host must employ careful validation of data that is passed over the privilege boundary. To avoid becoming a victim of a privilege escalation attack, the enclave must take special care to validate any input, particularly when the input contains code or data pointers.

As most C/C++ software, SGX enclaves are also prone to memory errors, which are the result of programming mistakes (as described in Section 2.1.1). However, in the context of the SGX technology, they are especially critical, as they allow an attacker to subvert all security guarantees provided by the hardware. For example, memory errors such as buffer overflows often allow an attacker to perform code-reuse attacks such as ROP [Sha07], or data-only attacks such as information leaks or data-oriented programming (DOP) [Hu+16].

Prior work has already analyzed exploit techniques for memory errors in the context of SGX enclaves. Lee et al. [Lee+17] presented DarkROP, a code-reuse attack technique, which shows that the enclave code must not be known to an attacker to successfully launch ROP attacks against the enclave. Biondo et al. [Bio+18] describes

a powerful exploit technique that allows the attacker to launch a code-reuse attack that abuses the particularities of the Intel SGX SDK. Furthermore, the code-reuse attack by Biondo et al. [Bio+18] bypasses existing randomization-based defenses such as SGX-Shield [Seo+17].

None of the prior works on SGX attacks has attempted to identify vulnerabilities in enclave code. All prior work assumed the presence of a memory error that can be abused by the attacker as a powerful memory corruption exploit primitive. Using our automated symbolic executor TeeRex, we shed some light on the prevalence of memory errors in enclave code. We gathered a set of seven enclaves, including open-source Linux enclaves and proprietary Windows enclaves. We find that in this set of seven enclaves, six contain critical vulnerabilities. Several of the enclaves, which we analyzed, are legacy code bases that have been ported to SGX enclaves. However, the software has not been updated to handle the particularities of the SGX environment. We conclude that due to memory errors and a lack of parameter validation on the host-enclave boundary, many SGX enclaves are vulnerable to privilege escalation attacks, such as leaks of enclave memory or even arbitrary code execution inside the enclave. To improve the security of SGX enclave code we 1) summarize common pitfalls of SGX code as vulnerability patterns (Section 4.2) and 2) present our analysis tool TeeRex to automatically vet enclave code for vulnerabilities (Section 4.1).

Contributions To summarize, the contributions of the work presented in this chapter are:

- We show how to adapt symbolic execution to SGX enclaves and exploit the parallelism inherent to the structure of the SGX APIs to speed up symbolic execution.
- We show how to detect memory corruption attacks in SGX enclaves using our adapted symbolic execution engine.
- We perform an in-depth analysis of several real-world enclaves using TeeRex. We discover critical vulnerabilities in all but one of the analyzed enclaves. We verify the vulnerabilities by constructing proof-of-concept (PoC) exploits based on the output of TeeRex.
- We present several vulnerability patterns that are specific to the SGX technology but common to many of the enclaves we analyzed. Along with the vulnerability patterns, we present guidelines to avoid said vulnerabilities.

This chapter is based on the following publication:
"TeeRex: Discovery and Exploitation of Memory Corruption Vulnerabilities in SGX Enclaves". ***29th USENIX Security Symposium*, 2020**. Tobias Cloosters, Michael Rodler, and Lucas Davi

4.1 The Symbolic Enclave Executor TeeRex

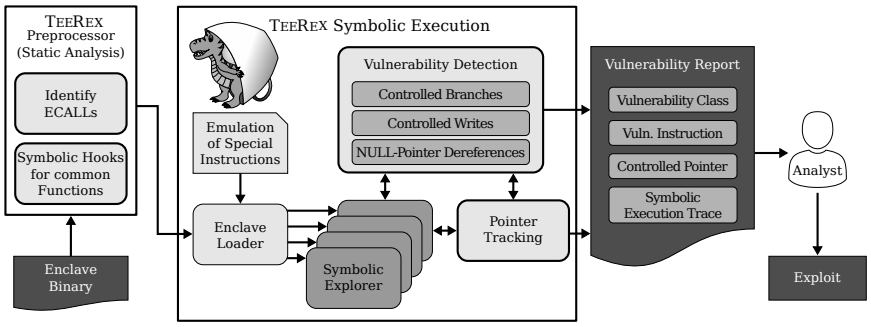


Figure 4.1: Architecture of TeeRex [CRD20]

To analyze SGX enclave code, we developed a symbolic execution tool. We call our symbolic executor TeeRex, an abbreviation of *Trusted Enclave Ecall Runtime EXploiter*. We use this symbolic execution framework to implement vulnerability detectors that *automatically* identify security-critical bugs in SGX enclave code. TeeRex generates a detailed vulnerability report, which allows a security analyst to quickly develop a proof-of-concept exploit that demonstrates the vulnerability. To facilitate proof-of-concept exploit development, TeeRex also discovers useful *exploit primitives*, i.e., code paths that are useful for constructing exploits.

TeeRex is built using the well-known Angr binary analysis toolkit [Sho+16]. By building upon Angr, we can easily support the major platforms that are also supported by SGX (more specifically the Intel SGX SDK): Windows (PE) and Linux (ELF) binaries and both 32 and 64-bit enclaves. In contrast to other symbolic executors, such as KLEE [CDE08], we apply symbolic execution to the *binary level*. This has several advantages: First, this allows TeeRex to analyze closed-source, proprietary enclaves where no source code is available. Second, the vulnerabilities we want to discover often depend on the low-level details of the executed enclave code. As such, binary-level symbolic execution is a natural choice. However, since we need to model several low-level execution details in TeeRex, we currently only support enclaves that utilize the official Intel SGX SDK. There are multiple other enclave frameworks that offer different programming models and ABIs that are currently not supported, e.g., the Graphene framework [TPV17] or Google's Asylo [Asy]. In principle, our approach can be adapted to such alternative frameworks, which we leave as future work.

Some frameworks, including the Intel SGX SDK, include support for dynamically loading and decrypting code into a running enclave. Fundamentally, any binary analysis technique must have access to the complete binary code to perform analysis. As such, TeeRex also supports only unencrypted enclave code.

In the remainder of this section, we describe the architecture of TeeRex (Section 4.1.1), several implementation challenges (Section 4.1.2), and finally, the details on the vulnerability detection components (Section 4.1.3).

4.1.1 Architecture

Applying symbolic execution to real-world software systems faces several challenges, including state explosion and interactions with other components. For example, dealing with side effects caused by the OS is highly challenging. Continuing symbolic execution into the OS kernel is possible [CKC11], but can easily lead to state explosion [Bal+18]. Realistically, to support symbolic system calls, a symbolic execution engine must simulate the system calls and manage a simulated file system [Bal+18]. Fortunately, many properties of SGX enclaves make it easier to perform symbolic execution for SGX enclaves: enclave code is completely self-contained, and the enclave is isolated from the rest of the system. In particular, SGX enclaves do not utilize dynamic libraries and cannot perform any system calls. Consequently, any interaction with the OS must be routed through the untrusted counterpart to the enclave. This is achieved by using the *OCALL* mechanism to the untrusted host application.

Figure 4.1 shows the architecture of our symbolic analysis in TeeRex. With TeeRex, we identify vulnerable program states while symbolically executing the enclave code. TeeRex also gathers meta-data along the way to generate a detailed vulnerability report. TeeRex symbolically executes each ECALL of the enclave and checks every resulting state for various vulnerability types. TeeRex contains a pointer tracking component used during vulnerability report generation. The pointer tracking component tracks pointer dereferences and propagates labels that allow TeeRex to distinguish data that originates from host and enclave memory. This is required because TeeRex needs to identify vulnerable code paths that load data from outside the protected enclave memory, i.e., untrusted host memory. Every load from untrusted host memory introduces a fresh symbolic value, allowing TeeRex to model arbitrary, and potentially malicious, host applications.

TeeRex is implemented based on the well-known Angr binary analysis framework [Sho+16] and utilizes the symbolic execution engines of the Angr framework. However, Angr itself does not fully support executing SGX enclaves out of the box since it does not model the division of the address space into a trusted and untrusted part. More specifically, (1) Angr cannot jump from the host application to the enclave (2) the initial environment to directly execute ECALLs requires special state setup, (3) some of the CPU instructions used by enclaves are not supported by Angr out-of-the-box, (4) Angr is limited to one thread and CPU core, while TeeRex is able to leverage enclave specifics (i.e., multiple ECALLs) to perform multi-core analysis, and (5) the functions that implement the trusted memory allocator are not directly supported by Angr. Furthermore, we extend Angr by adding vulnerability analysis components specific to SGX enclaves. Note that Angr does not perform any vulnerability analysis by default. It provides a robust framework to perform analysis and symbolic execution of binary code. We show how to tackle these challenges in TeeRex in Section 4.1.2.

TeeRex consists of several major components, as shown in Figure 4.1: the preprocessor, enclave loader, symbolic explorer, vulnerability detection, and pointer tracking component.

Preprocessor First, TeeRex performs a pre-processing step on the enclave binary to (1) identify instructions and functions that are not supported by the base symbolic executor Angr, and (2) to identify the ECALL table and extract its contents: the addresses of the ECALL functions. This first static analysis step is required to increase the performance and coverage of the analysis. Identifying the various ECALLs allows us to skip executing the initialization routines of the Intel SGX SDK. Furthermore, this also allows for parallelization, as different ECALLs can be analyzed in parallel.

Enclave Loader Before starting symbolic execution, we must first load the enclave code and set up an initial state inside of the symbolic execution engine. The enclave loader loads the enclave code and then selects one of the ECALLs and sets up the corresponding initial environment. Furthermore, it places hooks on several *common functions* and *special instructions*, which the preprocessor identified, such that these are emulated using Python code. Finally, it also creates the argument structure for the ECALL, which contains unconstrained and traceable symbolic values.

Symbolic Explorer The symbolic explorer is the component that performs symbolic execution utilizing the Angr framework. However, in contrast to plain symbolic execution, the ECALL centric exploration of TeeRex can be distributed across multiple cores and machines. The symbolic explorer analyzes each of the ECALLs individually. The results of all explorations are merged in the final vulnerability report, which is consumed by the analyst.

Vulnerability Detection During symbolic exploration, TeeRex' vulnerability detection components analyze the encountered program states. More specifically, the vulnerability detection components analyze all memory accesses (loads and stores) and all jumps. We describe the vulnerability detection components in greater detail in Section 4.1.3.

Pointer Tracking With TeeRex, we attempt to detect vulnerabilities at the interface boundary, which are caused by insecure pointer usage and lack of pointer validation. As such, we need to precisely model and track pointers during symbolic execution. To achieve this, we utilize a pointer tracking component that is hooked into the symbolic execution engine. We analyze all pointer dereferences and propagate labels between symbolic values. This is a taint-style analysis, where each value that is loaded from memory is also annotated with its address. In terms of taint analysis, the taint tag is equal to the concrete address where the value was loaded. We then propagate this during symbolic execution in TeeRex, such that when the memory-loaded value is used, we can determine the address where the value originated. With this information, we can track back the logical origin of the value. For example, when a function pointer is used as part of an indirect call, we can trace back whether the value was originally loaded from enclave memory, host memory, or whether it was passed as an argument to the ECALL function.

To further refine this distinction between trusted and untrusted memory, we place hooks on certain Intel SGX SDK functions that validate addresses. For example, there

are functions that validate whether an address is within or outside of enclave memory, respectively. When the enclave code utilizes such a function, TeeRex introduces two distinct follow-up states in the symbolic execution: one for within and one for outside enclave memory. When the enclave properly validates the pointer, then one of the two states will quickly lead to an error state, which is not further explored. This information gained by hooking these functions from the Intel SGX SDK is also used by TeeRex to check whether a bug is likely exploitable, which allows TeeRex to refine the vulnerability report's accuracy.

Vulnerability Report The final component of the analysis pipeline is the vulnerability reporting component. TeeRex produces a vulnerability report, which contains all findings ranked according to potential severity. For every finding, the report contains (1) the type of vulnerability, (2) the location in the binary (i.e., code address), (3) which pointer is controllable and the source of the controllable pointer, (4) and an execution trace that shows how vulnerable instruction can be reached. Based on this vulnerability report, an analyst can determine the severity of the findings, e.g., by constructing a proof-of-concept exploit.

4.1.2 Implementation Challenges

In the following, we will discuss the challenges of applying symbolic execution to SGX enclaves binaries and how we tackled these challenges with TeeRex.

Bypassing Initialization Code For enclaves built with the Intel SGX SDK, there is typically only one or only a few entry points into the enclave. For a typical ECALL, the enclave first executes setup routines from the TRTS, unpacks the parameters, and finally jumps to the ECALL handler in the enclave code (see Section 3.1 for more details). The setup routines in the TRTS are heavily dependent on the intrinsics and calling conventions of the SGX instructions and the enclave's internal metadata, which are not present in our emulated environment. This introduces a major challenge for a symbolic execution engine. First, the initialization routines contain many memory accesses through symbolic addresses, which is a notoriously hard problem for symbolic execution engines in general [Bal+18; Cha+12]. Second, due to the low-level nature of the TRTS code, the symbolic execution loses semantic information about the execution context when it finally reaches the ECALL handler. Due to this loss of semantic information, it is not easily possible to map symbolic memory ranges to ECALL parameters once the symbolic execution reaches the ECALL handler.

We are mostly interested in analyzing the ECALL handler code provided by the enclave developer. Everything between the enclave entry and the ECALL handler is code provided by the Intel SGX SDK. This code is unlikely to contain critical bugs, so we skip SDK code analysis. We designed TeeRex to skip the symbolic execution of the TRTS initialization and instead directly target developer-provided ECALL handler functions. In the pre-processing step, TeeRex first locates and extracts the ECALL table from the enclave binary, which maps an ECALL identifier to the ECALL handler function. TeeRex then starts symbolic execution at the beginning of every ECALL handler separately.

This design has several advantages: first, it allows TeeRex to produce very accurate vulnerability reports as it is now possible to map data that is used by the ECALL handler to arguments passed to the ECALL function. Second, it simply skips executing code that is not relevant for identifying vulnerabilities in enclave code, i.e., the TRTS code. Finally, it allows for a certain degree of parallelization because we can start the analysis for each ECALL function separately. This also allows us to deal with restrictions of the Angr framework, which is restricted to one thread due to the limitations of the Python implementation.

Standard Memory Functions Symbolically executing the binary code of standard helper functions such as `memcpy` or `malloc` quickly leads to unnecessarily complex symbolic program states and path constraints. This is because the symbolic execution engine must analyze all of the—surprisingly complex—internals of these functions. To avoid this unnecessary overhead, many symbolic execution engines hook such functions and implement symbolic summaries. The Angr framework employs so-called *SimProcedures* to implement such common functionality directly in the symbolic execution engine. For example, Angr hooks into the execution of `malloc`, and instead of symbolically executing the function in the binary or standard library, the *SimProcedure* is called. In Angr, the *SimProcedure* is typically implemented in python and updates the symbolic state in a similar way to the hooked function. However, in contrast to the original function, the *SimProcedure* has significantly reduced potential for state explosion. Typically, such helper functions are located in dynamically loaded system libraries such as the standard library for C (*libc*). These are easy to intercept, as they are identified by their name (e.g., the symbol in Executable and Linkable Format (ELF) binaries). In contrast to regular applications, SGX enclaves are always statically linked. Therefore, they ship with their own versions of these helper functions. It is not easily possible to identify the right place hooks, as no symbols are available. As such, TeeRex employs heuristics to search the enclave binary file for the statically linked versions of helper functions and places hooks to invoke the corresponding SimProcedure instead. Here we exploit the fact that most enclaves utilize the functions that are provided by the Intel SGX SDK.

Unsupported CPU Instructions SGX is a relatively new CPU extension, and it depends on several other newer extensions to the *x86* ISA. Unfortunately, several newer instructions are not supported by the version of the Angr framework on which TeeRex is based. As such, we need to work around this issue by placing hooks, similar to *SimProcedures*, on the unsupported instructions. We place hooks on the primary SGX instruction `enclu`, which is used to enter and exit an enclave. Additionally, we place hooks on the instructions `rdrand`, which is used in the enclave to generate cryptographically secure random numbers, and `xsave/xrstor`, which are used during OCALLs for saving and restoring the register state to/from memory when the execution passes the host-enclave boundary. Since we directly start symbolic execution within the ECALL, we do not need to support enclave entry. We stop analysis when we encounter an instruction to perform an enclave exit. For the other unsupported instructions, we simply implement their effects on the symbolic program state in the corresponding Python hook.

Global State of Enclaves and Chains of ECALLs A typical SGX enclave developed with the Intel SGX SDK can have multiple entry points. An attacker, i.e., the untrusted host application, can exercise the different ECALLs with different attacker-controlled input data in different orderings, with each call affecting the internal global state of the enclave (enclave heap, global variables, etc.). This is a typical API testing problem. As we discussed earlier in Section 2.2, exhaustive exploration of APIs using symbolic execution is often not feasible. Alternatively, an accurate symbolic exploration of an ECALL requires exhaustive knowledge about the effects of all ECALLs. However, obtaining the effects of an ECALL requires first knowing about the effects of other ECALLs. Typically such problems in program analysis are solved using fixed-point algorithms.

Instead of using a fixed-point iteration approach, we implemented TeeRex such that we sacrifice completeness and soundness for increased analysis performance. TeeRex analyzes each ECALL individually and treats all global state of an enclave as initialized with unconstrained symbolic values. This includes all global variables in the `data` and `bss` sections. Essentially, we treat the enclave's initial state similar to attacker input. While this makes TeeRex explore paths of an ECALL that are not reachable with an enclave's initial global state, it also produces false alarms. The reason for the increased number of false alarms is that an unconstrained global state is an over-approximation of the initial enclave's global state. The global state is typically not fully attacker-controlled, like ECALL inputs, but rather initialized to zero or set to some specific value by a different ECALL, according to the implemented enclave logic.

For example, consider an enclave that utilizes a function pointer stored in the global state. TeeRex assumes that this function pointer is unconstrained: allowing the enclave to jump anywhere. However, in reality, the jump may only target a certain small set of function pointers that are set by other ECALLs. Since TeeRex does not detect this implicit relationship, it will identify a jump that is controlled by the attacker. Nevertheless, the analysis results are still useful because they can lift limited exploitation primitives (e.g., null-pointer dereference or write to an arbitrary address with a fixed value) to full control-flow hijacking attacks. To increase usability, we introduce a classification of findings into auxiliary primitives and exploit primitives, where the former are not exploitable by themselves but are useful in constructing proof-of-concept exploits.

4.1.3 Vulnerability Detection Components

We implemented three major vulnerability detection components in TeeRex: (1) attacker-controlled branches (control-flow hijacking), (2) controlled writes, and (3) NULL-pointer dereferences.

Control-Flow Hijacking In TeeRex, we define control-flow hijacking as a program path where the symbolic execution engine encounters an unconstrained jump target. During regular executions, jump targets are always constrained to a very limited set of legitimate targets. However, suppose an untrusted attack-controlled pointer is used to determine a jump target. In that case, the jump target becomes unconstrained, as attacker-controlled input is set to unconstrained values in TeeRex.

TeeRex's symbolic execution engine sets anything that is potentially attacker-controlled to an unconstrained symbolic value. For example, if the enclave attempts to read memory outside of enclave memory, we return a fresh unconstrained symbolic value. Similarly, all ECALL arguments are set to unconstrained symbolic values. As such, if the enclave utilizes one of the ECALL arguments as a jump target, TeeRex will observe a jump to an unconstrained value. The same mechanism is also used when the enclave loads a function pointer from untrusted memory.

TeeRex also sets global variables inside the enclave to unconstrained symbolic value. The idea behind this is to simulate prior ECALLs and also detect uninitialized reads. However, this is an over-approximation, as not all global variables are necessarily attacker-controlled. We utilize the same detection mechanism to detect if the enclave attempts to use a value from uninitialized memory as a function pointer. However, we have to treat this finding differently during post-processing and vulnerability report generation.

If, on the other hand, the enclave includes validation code for the jump target pointer, then the symbolic execution will gather constraints on the symbol representing the jump target. The jump target is now constrained to be within a certain set of allowed—assumed to be safe—values, which will not trigger an alarm. We adapt the same convention as the underlying *Angr* framework and assume that a jump target is unconstrained if the SMT solver can determine more than 256 different concrete values for the jump target.

Controlled Write The second exploit primitive that TeeRex identifies are writes to arbitrary unconstrained memory addresses. An unconstrained memory write allows an attacker to overwrite critical memory, such as return address or function pointers, in enclave memory. To detect an arbitrary write, we utilize the pointer tracking component that tracks every pointer dereference and propagates labels similar to taint analysis [CLO07b; Vee+17]. This is needed to connect a pointer to its input source, including the level of pointer indirection and corresponding pointer offsets.

Whenever a pointer is utilized for a memory write, TeeRex checks whether the address is related to attacker-controlled memory. TeeRex then checks how the address is constrained. To make a useful exploit primitive, the memory write must be able to write into nearly-arbitrary enclave memory. TeeRex leverages the solver of the symbolic execution engine to solve the path constraints up to the memory write and query whether the address can possibly point to an arbitrary memory location within the enclave memory. If this is the case, then we report an arbitrary-write exploit primitive.

When analyzing the memory write, we focus primarily on the address. We consider it as an exploit primitive if the address is attacker-controlled. However, while we analyze the value that is written, we report a vulnerability regardless of the value. This is because any write, even of constant values, to an arbitrary address is very likely also exploitable in the SGX scenario. For example, even a controlled write with a fixed byte value is often sufficient to corrupt a pointer inside enclave memory. This is because the attacker has almost arbitrary control of the address space layout of the host application, including the enclave memory. As a result, it is often sufficient to

partially corrupt a pointer to point somewhere into insecure memory and simply map the resulting corrupted pointer into insecure memory. We describe one such exploit in Section 4.3. As such, TeeRex reports any memory write to an attacker-controlled address, regardless of the value written.

NULL-Pointer Dereference In the x86 virtual address space, the address valued 0 is a regular address and can be mapped. However, in C and C++, the null pointer is used to represent invalid pointers. For example, pointers are typically zero-initialized and, as such, point to the address 0 by default. Many libraries also utilize a null pointer to signal an error to the caller when a pointer is returned. For example, when a memory allocation with `malloc` fails, a null pointer is returned. As such, null pointer dereference bugs are relatively common in C/C++ code bases. However, null pointer dereferences are not considered especially critical in most settings. This is because the null page is not mapped in a typical user space process. In fact, many operating systems even prevent regular unprivileged processes from mapping the null page as they were historically often used to exploit null pointer dereferences in the operating system kernel. In stark contrast to other x86 software, in the SGX setting, a null pointer dereference bug is extremely critical and most certainly exploitable. The problem here is that the SGX enclave has nearly no control of the address space layout it is executed within. In the SGX setting, the operating system is also untrusted, so there is basically no way for an enclave to prevent a mapped null page. As such, TeeRex has a special vulnerability detection component that identifies null pointer dereferences. In TeeRex, we analyze every memory access (read or write) and check whether the address is pointing to the zero page mapped at address 0 (typically $<$ `0x1000`). Often, null pointer dereferences can also be captured by the other vulnerability detection components. However, an explicit detection component results in a better and more detailed vulnerability report.

4.2 Vulnerability Patterns

By design, SGX enclaves interface with untrusted code running in the same virtual address space, introducing two privilege levels that can easily share data. The privileged software, i.e., the SGX enclave code, can simply read the memory of the less-privileged software, i.e., the host application's memory. The hardware prohibits access in the opposite direction. Special care has to be taken when data is passed to the SGX enclave in the form of pointers, as any pointer passed to the enclave can naturally point to the whole address space, that is, the untrusted host and enclave memory. In this section, we discuss five major bug and vulnerability classes and several other minor vulnerability classes. We distilled these vulnerability classes from the findings that we presented in Section 4.3.

The common theme of all bugs we identified and discuss here is the *insufficient validation of pointers* at the host-enclave boundary. In the remainder of this section, we discuss the bug classes in detail and how they can be utilized during exploitation of enclaves. For every bug class, we also provide recommendations on how to avoid the vulnerability pattern.

4.2.1 Passing Data-Structures with Pointers

Most complex data types in C/C++ use pointers as a primary mechanism to form complex data structures like lists, trees, or maps. The Intel SGX SDK fully supports complex data types, including pointer-based data structures. However, currently, the Intel SGX SDK does not automatically perform a deep copy of pointer-heavy data structures. Consequently, it becomes dangerous to pass data structures containing pointers to an enclave. Especially when programming in C++, it is often unclear which data structures contain pointers and which do not.

Figure 4.2 shows an example of this pattern, where the enclave receives a linked list as an argument. Here each element of the linked list will point to the next element. The enclave receives the first element of the list as a parameter. This element is passed by value, which means that the Intel SGX SDK will automatically copy the first element to enclave memory. However, since the Intel SGX SDK does not recursively copy the list, only the first element of the list resides in enclave memory. The attacker can pass a list head structure that contains a `next` pointer pointing to arbitrary memory, including trusted enclave memory. When the enclave dereferences this pointer, the attacker can potentially corrupt enclave memory.

```c
// C Source
struct list_data;
struct list_head {
  struct list_head* next;
  struct list_data data;
  // ...
}
```

```
// EDL
enclave {
 trusted {
   void ecall(struct list_head list_root);
 };
};
```

Figure 4.2: Problematic EDL/C files which pass unsanitized pointers. The `next` pointer is not automatically validated and can point to arbitrary memory (including enclave memory).

Guideline: Any pointer passed to the enclave directly or as part of a data structure must be validated to point exclusively to normal world memory taking possible overlaps with enclave memory into account. Any data structure containing pointers must be treated the same way as pointers annotated with the `[user_check]` attribute. This means that in any entry point to the enclave (ECALL) the enclave must walk all data-structures and validate all pointers before the enclave dereferences any of those pointers. Due to possible TOCTOU issues, any data the enclave validates must be copied into enclave memory before validation. Furthermore, the data must be properly cleared if the validation fails to not retain untrusted data in enclave memory. The checks generated by the Intel SGX SDK only validate pointers non-recursively. Therefore, the generated checks are insufficient to validate complex data structures, such as lists, trees, or maps.

4.2.2 Using Pointers as Resource References

We observed the pattern that enclaves often provide the functionality to allocate some resource, e.g., a TLS session or file object. To identify and distinguish multiple resources, such as multiple TLS connections, the enclave returns a reference to this resource to the host application, which is then passed to further calls to the enclave. If the normal world code wants to use the newly allocated resource, then the corresponding function of the enclave receives the corresponding reference as a parameter. In C/C++ code, this is typically achieved by returning and passing a pointer to the object containing the resource's data. Especially ports of legacy code bases to SGX retrain this pattern in the enclave code. However, in SGX enclaves, this is an extremely dangerous coding pattern. It is not possible for an enclave to properly validate the pointer passed to the enclave.

We observed that enclaves perform some input validation in this case. They will typically perform a check to validate that (a) the pointer is not null, (b) and the pointer is pointing to enclave memory. However, it is fundamentally impossible for the enclave to validate whether the pointer is really pointing to an object with the right type. As such, it is possible for the attacker to violate the type safety of the enclave by passing a pointer to some enclave memory. In turn, this will very likely lead to memory corruption, where the attacker abuses the regular processing of the enclave to corrupt enclave memory.

As we show in Section 4.3, this pattern is typically exploitable. In many enclaves, it is possible for an attacker to control some content of the enclave memory simply by providing input to some enclave function. The trusted runtime part of the enclave then copies the input (e.g., a simple string) into enclave memory.

Furthermore, it must be assumed that an attacker can break any information-hiding defense, such as address or code randomization. As such, the attacker knows the address of any attacker-controlled input in enclave memory [Lee+17]. This gives the attacker all necessary prerequisites to inject a fake object into enclave memory. Tricking the enclave into using this injected fake object will typically lead to memory corruption.

Figure 4.3 shows a typical example of this problematic code pattern: A session pointer is returned to the normal world in the `new_session` function and then supposed to be passed to other ECALLs, such as the `close_session` function. However, the enclave cannot easily validate the passed session pointer, leading to memory corruption when an attacker passes a pointer to some other arbitrary enclave memory.

Guideline: Enclaves must never return pointers to enclave memory to the host application and must never take a pointer, which may point to enclave memory, as input from the host application. Due to a lack of type and memory safety on the host-to-enclave boundary, an enclave cannot validate pointers passed to the enclave based on the pointer alone. Whenever an enclave must return a reference to an object in enclave memory to the host application, the enclave must protect the reference. We suggest allocating an array of pointers to enclave objects inside of enclave memory and returning array indices to the host application instead of pointers, as shown in Figure 4.4. This approach is similar to the use of file descriptors on Unix-like systems.

```
// EDL
enclave {
 trusted {
  struct Session*
  ↪  new_session();
  void close_session(
   [user_check] Session* s
  );
  /* ... */
 };
};
```

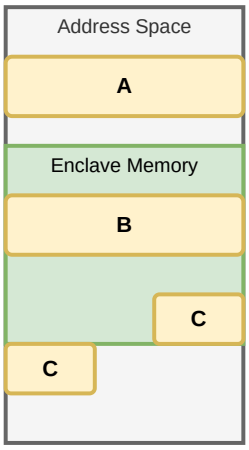
```
// C++ Source
struct Session { /* ... */ }
Session* new_session() {
  // allocate new Session on enclave heap
  return new Session();
}
/* ... */
void close_session(Session *s) {
  // insufficient pointer validation
  const size_t SZ = sizeof(Session);
  if (!sgx_is_within_enclave(s, SZ))
  {
    return;
  }
  // possible memory corruption
  delete s;
}
```

Figure 4.3: Example of enclave code passing a pointer to/from untrusted user space as a resource reference, a session object in this case. The pointer validation in the `close_session` function is insufficient and can lead to memory corruption, when an attacker passes a bogus session pointer.

```
// EDL
enclave {
 trusted {
  uint32_t new_session();
  void close_session(
   uint32_t session
  );
  /* ... */
 };
};
```

```
// C++ Source
struct Session { /* ... */ }
uint32_t next_session = 0;
struct Session sessions[MAX_SESSIONS] =
↪ {nullptr,};
uint32_t new_session() {
  if (next_session < MAX_SESSIONS) {
    uint32_t r = next_session;
    sessions[r] = new Session();
    next_session++;
    return r;
  }
  return -1;
}
/* ... */
void close_session(uint32_t session) {
  if (session < next_session) {
    if (sessions[session]) {
      delete sessions[session];
      sessions[session] = nullptr;
    }
  }
}
```

Figure 4.4: Using a session identifier instead of a pointer to avoid passing pointers across privilege boundaries.




```
1  sgx_is_outside_enclave(A, sz) == true
2  sgx_is_within_enclave(A, sz)  == false
3
4  sgx_is_outside_enclave(B, sz) == false
5  sgx_is_within_enclave(B, sz)  == true
6
7  sgx_is_outside_enclave(C, sz) == false
8  sgx_is_within_enclave(C, sz)  == false
```

Figure 4.5: Considering three different scenarios of buffer locations: Buffer `A` is located strictly outside enclave memory, `B` strictly within and buffer `C` is neither. When validating pointers the enclave must consider the scenario of overlapping buffers.

4.2.3 Pointers to Overlapping Memory

We observed that when passing pointers between the enclave and the normal world, the enclave often utilizes the functions provided by the Intel SGX SDK to validate the given pointer. One important aspect of pointer validation is to check whether the object behind the pointer is contained within the enclave memory or outside. The Intel SGX SDK provides two functions to achieve such validation: `sgx_is_within_enclave` and `sgx_is_outside_enclave`. Both functions receive a pointer and a length and return a boolean flag, whether the given object, as defined by pointer and size, is *strictly* outside or within enclave memory. However, there are three different scenarios that must be covered by the pointer validation code: (1) stricly inside, (2) strictly outside, (3) overlapping. Figure 4.5 shows a visualization of the three different scenarios and corresponding calls and results of the validation functions from the Intel SGX SDK.

However, this API design can lead to unexpected results if the user is not aware of the three scenarios. Figure 4.6 shows an example of a bug introduced by wrongly assuming the negation of `sgx_is_within_enclave` is equal to calling `sgx_is_outside_enclave`. The edge case of overlapping memory was not considered in this example. The developer's intention was to validate the pointer by checking whether the provided memory area is inside of enclave memory and return an error if it is not. However, the pointer validation with `sgx_is_outside_enclave` in Figure 4.6 can be bypassed by an attacker. They pass a memory object starting inside of enclave memory but extending into non-enclave memory.

We observed this pattern (see Section 4.3) in conjunction with a previous pattern (see Section 4.2.1). As such, when following the guidelines of this section, an enclave should never have to validate a pointer parameter to point inside enclave memory. However, the same issue also applies to pointers passed to the enclave as *output parameters*. For example, if the enclave decrypts some content on behalf of the host

```
// buggy check
if (sgx_is_outside_enclave(addr,
    size))
{
  return ERROR;
}
// now addr still may be partly
    outside
```

```
// correct check
if (!sgx_is_within_enclave(addr,
    size))
{
  return ERROR;
}
// now addr is strictly within
// the enclave memory
```

Figure 4.6: Enclave code that tries to validate that a memory region is inside of enclave memory, but fails to handle the edge-case of memory overlapping between normal-world and enclave memory. A proper check would validate whether the memory area is not within enclave memory.

program and writes the result back to host memory via a pointer received as an argument. The enclave must validate that the output pointer does refer to a memory area fully outside the enclave memory.

Guideline: The enclave must validate whether a pointer references normal or enclave memory. We recommend to restrict pointer validation code to usage of the function `sgx_is_outside_enclave`. This allows the enclave to validate that any buffer is strictly outside. Usage of the function `sgx_is_within_enclave` should be unnecessary when adhering to the guideline with respect to passing pointers as resource identifiers (see Section 4.2.1).

4.2.4 NULL-Pointer Dereferences

C/C++ code commonly uses the special *NULL* (or *nullptr*) value to signal that a pointer is not initialized or has been cleared. In reality, the pointer is simply set to the numeric value 0. However, on a typical x86 system, which utilizes virtual memory, the address 0 is a valid address. Typically, there is no valid memory located at the address 0. As such, any accidental dereference of a null pointer results in a crash of the process (i.e., a *SEGFAULT*).

However, in contrast to regular user space software, null pointer dereferences become more dangerous in privileged software. For example, a common source of kernel vulnerabilities in most major operating systems was null pointer dereferences. The address 0 is part of user space memory in most major OS designs. Therefore, a null pointer dereference can be exploited by a malicious process that maps valid memory at address 0. In this case the null pointer dereference turns into a valid pointer dereference. When the kernel attempts to load a value through a null pointer, the attacker can then make the kernel load a bogus value from the page at address 0 instead of crashing. As mitigation for this type of attack, many modern OS kernels, such as Linux or the Windows kernel, disallow mapping any memory at address 0. Furthermore, many CPU architectures introduced special CPU modes that prevent privileged kernel code from accidentally accessing user space memory.

Similarly to the kernel, null pointer dereferences in SGX are more dangerous than in normal user space. In contrast to kernel code, there is no mitigation available for

null pointer dereferences inside SGX enclaves. First, the enclaves are considered part of the user space application. Second, the OS kernel is untrusted in the SGX threat model. As such, it must be assumed that the OS kernel is under the control of the attacker, and the attacker can freely map the page at address 0. Since SGX enclaves cannot directly jump into normal world code, they cannot jump to address 0 mapped into normal world user space. As such, a null pointer dereference while performing a jump or call is not exploitable. However, other null pointer dereferences must be considered an attack vector for data-oriented attacks [Hu+16] in the SGX enclave. Especially problematic are null pointer dereferences that occur when loading code pointers. These give the attacker control over the control flow of the enclave code, facilitating code-reuse attacks.

Guideline: An enclave must never dereference a null pointer. Any pointer dereference of a nullable reference must be guarded by a null pointer check. Furthermore, we recommend utilizing C++ references, which are essentially non-nullable pointers. They cannot result in a NULL pointer dereference since they are required to be initialized with a non-null value.

4.2.5 Time-of-Check Time-of-Use

Enclaves in SGX run as part of a normal x86 user space process. As such, enclave developers must ensure that the enclave code is also secure with respect to concurrency bugs (see Section 2.1.2 for a general introduction to concurrency bugs). Intel already included a method to limit concurrency for enclaves. The enclave developer can specify how many concurrent threads may call into the enclave. This is done by setting the number of TCS entries in the enclave's metadata. As such, a thread-unsafe enclave can be made accessible only to one host application thread at a time.

However, enclaves must also be aware of TOCTOU issues during input validation. When an enclaves accesses host application memory, the enclave must assume that a separate host application thread can change the content in the untrusted memory area. This problem is amplified by the fact that a malicious host can interrupt and pause enclave threads at any time, almost at the instruction granularity [VPS17].

Figure 4.7 shows an example of such a bug, where the enclave attempts to traverse a single-linked list in host application memory. When the enclave utilizes the *next* pointer to traverse to the next list entry, the enclave code validates that the *next* pointer does not refer to an object in enclave memory. However, the depicted code in Figure 4.7 is vulnerable to a TOCTOU issue due to a double-fetch bug. Here, the *next* pointer is always fetched from normal world memory. First, the enclave validates that it does not point to enclave memory with the `sgx_is_outside_enclave` function. After the call to the validation function, the code fetches the *next* pointer again from untrusted memory and assigns it to the *head* variable. However, a separate host application thread can easily change the value of the pointer *next* between the call to the validation function and the assignment to the *next* pointer. Effectively, this allows an attacker to bypass the pointer validation.

```
1  struct X {
2    struct X* next;
3    // ...
4  }
5  struct X* head = some_normal_world_memory; // FETCH
6
7  // TIME-OF-CHECK
8  if (sgx_is_outside_enclave(head->next), sizeof(struct X)) {
9    // TIME-OF-USE
10   head = head->next; // DOUBLE-FETCH
11   // ...
12 }
```

Figure 4.7: SGX pointer validation vulnerability due to a time-of-check time-of-use problem, specifically due to a double-fetch from untrusted memory.

Guideline: Enclaves must not operate on data structures located in untrusted memory. Enclaves must copy data structures into enclave memory before validation.

4.2.6 Minor Vulnerability Patterns

We identified several more potential vulnerability patterns during analysis of the attack surface of the interfaces of enclave developed with the Intel SGX SDK. However, we did not find any real-world enclave that exhibits the following code patterns. As such, it is still an open question whether any of the following patterns are common in enclave code. However, for the sake of completeness, we include a detailed description of the patterns. As they are similar to bugs found in non-enclave code, we believe it is likely that inexperienced enclave developers could introduce one of the following issues.

Use-After-Free of ECALL Parameters

Many of the previously discussed patterns are due to developers working around the limitations of the Intel SGX SDK using data structures and user-checked pointers. However, there are also pitfalls when dealing with the auto-generated code of the Intel SGX SDK. More specifically, the SDK generates code for copying primitive data types between host and enclave memory on entry or exit of ECALLs and OCALLs. For example, the SDK fully supports the C string type, i.e., a zero-terminated array of characters. Here the SDK will automatically include code in the enclave to determine the string length in host memory, allocate memory in the enclave for the string, and copy the contents of the string from host memory to enclave memory. The custom ECALL handler of the enclave developer then simply receives a *char* pointer as an argument like an ordinary C function. Similarly, any data structure of fixed size is copied to enclave memory and passed by reference to the ECALL handler.

There is a potential pitfall with respect to the lifetime of ECALL parameters that are passed by reference to the ECALL handler. Namely, the lifetime of the object is the same as the duration of the ECALL. Essentially, the trusted runtime of the Intel SGX SDK performs an allocation before calling the ECALL handler and frees the

```
// EDL
int first_ecall([string,in] char* x);
void second_ecall([in] struct some_t* t);
/*****************************************************/
// C Source Code
char* global;

int first_ecall(char* x) {
  // keep reference in global variable (.data or heap)
  global = x;
  /* ... */
  // after the return, global becomes a dangling pointer
}

void second_ecall(struct some_t* t) {
  // use global in a second ECALL: UAF
  // global -> now points to some other object
  memcpy(global, /* ... */);
}
```

Figure 4.8: Use-After-Free (UAF) when the second ECALL uses a shared pointer variable, set by the first ECALL to a parameter, which is not valid after the first ECALL returns.

object after the ECALL returns. As such, the parameters must be basically treated as local variables with automatic lifetime. However, as they are passed by reference, it is easy to confuse these parameters with objects that have a global lifetime and leak references into global variables or heap-allocated data structures. Effectively, if a pointer to an ECALL parameter is stored in global enclave state, the pointer becomes a dangling pointer at the end of the ECALL. A second ECALL, will then very likely reuse the memory for the parameter, potentially giving the attacker access to a UAF issue, where the dangling pointer now points to an attacker-controlled memory area. Figure 4.8 shows an example that illustrates this issue. The first ECALL handler assigns the string parameter to a global variable, while the second ECALL handler then utilizes this global pointer.

Guideline: Any ECALL parameter must be treated as a local variable and must never be stored in global enclave state, such as global variables or in heap-allocated data structures. In general, such lifetime issues can often be detected using static analysis tools. However, to detect this pattern, the analysis tool must be adapted to the Intel SGX SDK generated enclave code, as the local lifetime of the ECALL parameters is not directly visible to an analysis tool.

Explicit Length vs. 0-Termination

C-style strings are notorious for causing memory corruption bugs during string handling due to the fact that their length is determined by appending a zero-byte termination. Most string processing functions simply receive a pointer to the beginning of the string

```
1  // EDL
2  int ecall_func([in,string] char* x, size_t len);
3  /*******************************************/
4  // Source
5  int ecall_func(char* x, size_t len) {
6    // ...
7    func(x, len);
8    // ...
9  }
10
11 int func(char* c, size_t len) {
12   // ....
13 }
```

Figure 4.9: Legacy C function ported to an SGX enclave.

without knowing whether the given string is heap-, stack- or statically allocated. As such, string parameters cannot be grown automatically, leading to buffer overflows if a function attempts to modify a string in a way that would require increasing the length of the string. Similarly, enclaves that receive string parameters must be careful to properly handle the string and not introduce buffer overflows.

To avoid trouble with the C-style strings, many C code bases utilize explicit length parameters to determine the length of strings instead of relying only on the zero termination. This allows functions to perform bounds-checking with respect to the underlying allocation of the C string. However, if such a function is ported to an SGX enclave, it is easy to introduce a problem as depicted in Figure 4.9. Here, a legacy C code base was ported to an SGX enclave. The enclave exposes an ECALL that receives a string parameter and a length parameter. The ECALL handler wraps an internal function *func* that receives a buffer and its length. However, in this case, the wrapper code generated by the Intel SGX SDK will handle the string parameter differently. More specifically, the SDK will generate code to determine the string length using zero termination and copy this length to enclave memory. However, the SDK will not consider the *len* parameter as the actual length of the buffer. Instead, the *len* parameter can be chosen arbitrarily by the attacker. For example, the attacker could choose a length parameter that exceeds the size of the string buffer, which causes a buffer overflow during processing.

We found short code excerpts containing this vulnerability in the official developer reference documentation of the Intel SGX SDK for Linux. However, we did not encounter functional enclaves that exhibit this issue.

Guideline: Enclaves must always rely on the Intel SGX SDK generated code to determine the length of buffer and string parameters. Length parameters must always be marked explicitly in the EDL file for the Intel SGX SDK to consider.

```
1  // EDL
2  int ecall_strcat([string,in,out] char* dest,
3                   [string,in]     char* src);
4  /*****************************************/
5  // Source
6  int ecall_strcat(char* dest, char* src) {
7    // buffer overflow inside of enclave memory!
8    strcat(dest, src);
9  }
```

Figure 4.10: Vulnerable use of out-strings.

Reusing Input Strings as Output

The C language does not allow for larger return values. To work around this limitation and to avoid needless copying, output parameters are used. The C function receives a pointer as an argument that is then used to store the return value. Similarly, SGX enclaves built with the Intel SGX SDK support output parameters for ECALL handlers. More specifically, in the EDL file, the developer tags the pointer parameter with `[out]`.

However, there is a problem if an enclave reuses a single parameter as an input and output parameter. This problem is illustrated in Figure 4.10, where the example enclave receives a string parameter that acts both as an input and output parameter. The enclave then concatenates a second input string to the output parameter. However, the Intel SGX SDK automatically determines the input string length and creates a copy in enclave memory As a result, the length of the output parameter is implicitly limited to the length of the input parameter. Appending data to the output parameter results in a buffer overflow within enclave memory, even if the respective buffer in host memory is large enough to store the output string.

Guideline: Enclaves should use dedicated separate arguments for the input and output of strings.

4.3 Enclave Analysis Results

We gathered a set of 7 publicly available enclaves to evaluate the effectiveness of TeeRex. We find that TeeRex is capable of identifying memory corruption vulnerabilities at the host enclave boundary in six out of the seven enclaves. Note that our dataset consists of real-world enclaves, including open-source and proprietary enclaves. This shows that memory corruption errors at the interface boundary are common in current SGX enclaves.

We give an overview of the analyzed enclaves in Table 4.1. We gathered several enclaves that are part of the official documentation of larger projects: the Intel GMP Example [SGXGMP] developed by Intel as an official showcase for the Intel SGX SDK, the Rust SGX SDK's tlsclient [Dua+; Wan+19a] developed at Baidu as part of their efforts in developing a Rust-based version of the Intel SGX SDK, and the WolfSSL Example Enclave [WOLFex], which shows how to utilize WolfSSL in SGX.

Furthermore, we include the academic project TaLoS [Aub+17; TaLoS]. We also analyzed the SignalApp Contact Discovery [Mar17] that is part of the Signal messenger ecosystem. All of the above enclaves have source code available. To demonstrate the effectiveness of TeeRex on enclaves without available source code, we also analyze two proprietary enclaves that are used by fingerprint drivers: Synaptics SynaTEE Driver and Goodix Fingerprint Driver. These enclaves have been deployed to Dell and Lenovo laptops.

To analyze the enclaves in our dataset, we first run TeeRex on the enclave. We then analyze the vulnerability report produced by TeeRex. If source code is available, we utilize the source code to perform root cause analysis and map the root cause to one of the patterns described in Section 4.2. If no source code is available, we cannot always map the results to one of the vulnerability patterns, as we have to resort to reverse engineering. Based on the vulnerability report, we then construct PoC exploits to verify whether the finding is truly exploitable. In our PoC exploits, we typically aim to hijack the instruction pointer inside of the enclave. With control over the instruction pointer, we can then rely on existing code reuse techniques, such as ROP [Sha07], to further attack the enclave. For example, we applied the code-reuse technique introduced by Biondo et al. [Bio+18], which targets enclaves built with the Intel SGX SDK.

We assume the standard SGX threat model [CD16; McK+13] for our PoC: our PoC exploits assume they have full control over the OS and the full user-space address space. Some of our PoCs exploits need to map data at address 0 to exploit *NULL* pointer dereferences. To achieve this, we configure the Linux systems, based on Ubuntu 18.04, to allow this. For the Windows enclaves, we had to resort to live patching the Windows 10 kernel to allow mapping the *NULL* pointer. For the sake of simplicity of our PoCs we disable address space layout randomization (ASLR), which is well within the capabilities of the attacker.

Notably, the SignalApp Contact Discovery enclave is the only enclave where TeeRex did not identify a vulnerability. We found that this enclave has a comparatively small and simple ECALL interface. As such, there is less possibility for mistakes. We can only speculate, but we assume that this is because the enclave was developed by

Table 4.1: Dataset of public enclaves and their susceptibility to exploitation.
* One ECALL immediately branches to 75 different actions, which we model as separate ecalls in TeeRex.

Project Name	Analyzed Version	Exploit	Patch Available	Source Code	Number of ECALLs
SignalApp Contact Discovery	1.13	×	-	✓	7
Intel GMP Example	`9533574f95b97`	✓	✓	✓	6
Rust SGX SDK's tlsclient	1.0.9	✓	✓	✓	8
TaLoS	`bb0b61925347b`	✓	×	✓	207
WolfSSL Example Enclave	`d330c53baff52`	✓	✓	✓	22
Synaptics SynaTEE Driver	5.2.3535.26	✓	✓	×	2 (76)*
Goodix Fingerprint Driver	2.1.32.200	✓	✓	×	56

the security and privacy specialists of the Signal Foundation. For all the identified vulnerabilities, we performed responsible disclosure and worked with the vendors to fix the discovered issues. In the remainder of this section, we discuss the technical details of the findings.

Table 4.2: Overview of results of our analysis of public enclave code. Some of the patterns are not applicable to some of the enclaves either, because the enclaves does not use the relevant code constructs, or the source is unavailable and thus we cannot determine the intention.

Bug Classes	Intel GMP Example	Rust SGX SDK's tlsclient	TaLoS	WolfSSL Example Enclave	Synaptics SynaTEE Driver	Goodix Fingerprint Driver
P1: Passing Data-Structures with Pointers	●	●	●	-	●	●
P2: Returning pointers to enclave memory	●	●	●	●	-	-
P3: Pointers to Overlapping Memory	-	●	-	-	-	-
P4: NULL-Pointer Dereferences	-	-	●	-	●	●
P5: Time-of-Check Time-of-Use	-	-	●	-	-	-
Exploit Primitive						
Control-Flow Hijack	-	●	●	●	●	●
Controlled Write	●	-	-	-	●	●
NULL-pointer Dereference	-	-	●	-	●	●

Intel GMP Example

As part of the documentation on SGX Intel provides a demo enclave that showcases the support of the *GNU Multiple Precision Arithmetic Library*. The whole enclave features a bad design and contains several insecure code patterns. Most notably the enclave freely passes pointers between the enclave and host boundary. TeeRex also identifies several arbitrary write exploit primitives. However, they all share the same root cause. As such, we now only describe one vulnerability in more detail.

The enclave features an ECALL, which wraps the addition functionality of the GMP big integer library. This ECALL receives three GMP big integers as parameters: two input parameters for the addition and an output parameter. The enclave adds both input parameters and then copies the result back to the output parameter. All parameters are pointers that are annotated with the *user_check* attribute, which means that the developer must validate these pointers.

TeeRex identified an arbitrary write exploit primitive in the enclave code, which we then turned into arbitrary code execution as part of our PoC exploit. Note that while the use of the *user_check* attributes already hints at a vulnerability, the actual problem discovered by TeeRex is a pointer within the passed data structure. As such, even if the data structure was changed to be fully copied to enclave memory, the issue would persist. The problem is that the data structure behind the GMP big integer internally utilizes a pointer to refer to an underlying buffer. The GMP big integers consist of several limbs, or machine native words, which are stored in this buffer. TeeRex identifies this pointer to the backing buffer as an attack vector. This pointer is not sanitized, which means that during the *mpz_set* operation, a memory write to an arbitrary location is possible. Interestingly, this vulnerability showcases the danger of utilizing an opaque data structure coming from a software library at the host-enclave boundary. It is easy to miss that such a data structure actually contains a pointer that must be sanitized.

The big integer data structures utilize dynamically allocated storage internally. As such, they must contain a pointer to the underlying buffer that stores the values of the limb integers. Figure 4.11 shows part of the vulnerable code. Here, the `mpz_set` function simply copies the output to the attacker-controlled underlying buffer of the `c_unsafe` big integer. The problem is that the enclave uses functionality of the GMP library that was not designed for SGX. It neglects the fact that the underlying buffer of this big integer can actually point to arbitrary memory, including enclave memory.

This vulnerability allows an attacker to perform an arbitrary memory write with controlled content and controlled size. In our PoC exploit, we abuse the `e_mpz_add` ECALL. We utilize the input parameters to control the values that are written. We set `a_unsafe` to the memory contents that we want to write. We set the big integer `b_unsafe` to a big integer initialized as 0, ensuring that the actual computation of the GMP library has no effect. We then manipulate the pointer to the underlying buffer of the `c_unsafe` data structure to point to our target address for the arbitrary write. We choose to directly write to the enclave stack, which allows us to directly write a ROP payload to the enclave stack. Once the enclave returns, it will execute our ROP payload.

```
// EDL
public void e_mpz_add(
  [user_check] mpz_t *c_unsafe,
  [user_check] mpz_t *a_unsafe,
  [user_check] mpz_t *b_unsafe
);
```

```
void e_mpz_add(mpz_t *c_unsafe,
               mpz_t *a_unsafe,
               mpz_t *b_unsafe) {
  mpz_t a, b, c;

  // ...
  /* The enclave now computes: */
  /* c = a + b                 */
  // ...

  // mpz_set copies the underlying buffer
  // of the biginteger "c" to the buffer pointer
  // contained in the "c_unsafe" variable
  mpz_set(*c_unsafe, c);
}
```

Figure 4.11: Excerpt of the vulnerable code in the *Intel GMP Example* enclave.

Intel acknowledged the problem, updated their documentation, and fixed the issue by using serialization. Instead of passing pointers to GMP structures, the demo code now serializes GMP big integer objects to strings. This avoids dangerous pointer validation as strings are automatically validated by the Intel SGX SDK. Inside the enclave, the input parameters are deserialized, the computation is performed, and then serialized and returned to the host application. This completely avoids the problematic pattern of passing pointers between host and enclave.

WolfSSL Example Enclave

The *WolfSSL* [WOLF] project develops a TLS library that is also fit for other use cases, such as embedded devices and small and self-contained applications. As such, the library comes without the need for external dependencies. This also makes the WolfSSL library applicable to run in the SGX context. The library ships with an example for utilizing the WolfSSL library code within a SGX enclave. This example shows how to terminate TLS connections directly inside the SGX enclave, shielding the cryptographic secrets required for running the TLS protocol from the OS. However, to achieve this, the enclave exposes many APIs, which essentially wraps the original WolfSSL API.

An analysis with TeeRex revealed a control-flow hijacking exploit primitive. Our root-cause analysis showed that the enclave follows the dangerous pattern of passing pointers and performs insufficient validation (see Section 4.2). More specifically, the

```
/* ECALL Definition in EDL */
// a pointer to enclave memory returned
public WOLFSSL* enc_wolfSSL_new([user_check] WOLFSSL_CTX* ctx);
// pointer is passed to enclave
public int enc_wolfSSL_connect([user_check]WOLFSSL* ssl);
// ...
```

```
/* C Source Code */
typedef int (*CallbackIOSend)(WOLFSSL *ssl, char *buf,
                              int sz, void *ctx);
/* WolfSSL session type */
struct WOLFSSL {
  WOLFSSL_CTX*    ctx;
  /* ... */
  // attacker-controlled function pointer!
  CallbackIOSend  CBIOSend;
}
// ...
int enc_wolfSSL_connect(WOLFSSL* ssl) {
  // insufficient validation
  if(sgx_is_within_enclave(ssl, wolfSSL_GetObjectSize()) != 1)
    abort();
  /* ... */
}
```

Figure 4.12: Relevant parts of the EDL definition and C source code of the *tlsclient* enclave.

enclave API requires the user to allocate a TLS session object in secure memory before calling any other APIs. The session object is represented as a raw pointer and returned by the enclave to the host application. The host application then passes the raw pointer to the enclave in subsequent calls, e.g., when data is received over the network. The enclave uses the same pattern also for other types, such as a *context* type or I/O buffer objects. We discuss one of the exploit primitives discovered by TeeRex in more detail.

Figure 4.12 depicts the vulnerable code of the enclave. To allocate a new TLS session, the host application passes a *WOLFSSL_CTX* pointer type to the enclave. The enclave returns a pointer to the *WOLFSSL* object that represents a TLS session. TeeRex identifies a weak point in this design: the `WOLFSSL` data structure contains a function pointer that the enclave uses to perform callbacks in the TLS library (*CBIOSend*). The usage of this function pointer in the *WOLFSSL* data structure is discovered by TeeRex as a control-flow hijack, as the pointer to the data structure is fully attacker controlled.

However, the enclave does not accept fully arbitrary pointer parameters. The enclave code validates that the pointer passed in the ECALL does in fact point to enclave memory. However, this pointer validation is not sufficient to prevent the exploitation. As discussed in Section 4.2, the enclave memory often contains attacker-controlled

content. For example, if the enclave attempts to validate data passed by host memory or if the enclave accepts string parameters. In our PoC exploit, we abused the function *enc_wolfSSL_CTX_use_PrivateKey_buffer* to inject a fake `WOLFSSL` data structure into enclave memory. However, there are multiple other suitable APIs. Our PoC abuses the *enc_wolfSSL_connect* function, which retrieves the function pointer for the callback from the injected data structure, giving the attacker full control over the instruction pointer. Ultimately, this allow the attacker to perform code-reuse attacks and exfiltrate any data that is supposedly protected.

We disclosed this issue to the authors of the WolfSSL project. Similar to our recommendations in Section 4.2, they switched to using integer-based session identifiers instead of passing pointers.

Rust SGX SDK's tlsclient/server

The *Rust SGX SDK*, developed at Baidu and later moved to the Apache Project, is an interesting analysis target. This project attempts to build a memory-safe wrapper around the Intel SGX SDK, using the *Rust* language as a modern memory-safe alternative to C/C++. As such, enclaves built using this SDK contain fewer memory safety issues. However, our analysis revealed memory safety issues at the boundary between the host and enclave: an interface that is not covered by Rust's guarantees. More specifically, the programmer has to resort to the C ABI to communicate between host and enclave. In turn, this makes it necessary to use *unsafe* code in rust, code where the compiler does not guarantee memory safety.

We analyze code shipped with the Rust SGX SDK that shows how to run a TLS server and client inside an SGX enclave. In terms of functionality, this is quite similar to some of the other enclaves we analyzed, e.g., the *WolfSSL* or *TaLoS* enclaves (see Section 4.3 and Section 4.3 respectively). The Rust SGX SDK project ships with two applications that represent a TLS client and server, which allows direct communication between two enclaves. Since both applications are quite similar with respect to the enclave interface, we only analyzed the *tlsclient* enclave.

Similar to the other TLS enclaves, the rust enclave exposes an enclave API that exposes a function to create a new TLS session. This session object can then be used to encrypt and decrypt data that is sent or received. TeeRex identifies a vulnerability leading to instruction pointer control in the `tls_client_write` function. Here the exploit primitive discovered by TeeRex abuses the session pointer parameter of the ECALL. The enclave utilizes the common anti-pattern of passing a pointer as a resource identifier. TeeRex then identifies a way to exploit this pattern.

Figure 4.13 shows the vulnerable part of the *tlsclient* enclave code. Here, a new TLS session object is allocated with the *tls_client_new* function. Subsequently, the returned pointer must be passed to further API calls, such as the *tls_client_write* function depicted in Figure 4.13. The parameter must be marked as *user_check* for the Intel SGX SDK to accept a raw pointer. This moves the burden of validating a pointer to the enclave developer.

TeeRex identifies a nested object inside the *TLSSession* data structure, which contains a virtual method table (vtable) pointer. This vtable pointer is used by the compiler to implement dynamic dispatch, i.e., it associates a set of method

```
/* ECALL Definition in EDL */
public void* tls_client_new()
public int tls_client_write(
    [user_check]   void* session,
    [in, size=cnt] char* buf,
                   int cnt);
```

```
// Rust Source Code
pub extern "C" fn tls_client_write(session: *const c_void,
                                   bu: * const c_char,
                                   cnt: c_int)  -> c_int {
 ① if session.is_null() {
        return -1;
    }

 ② if rsgx_raw_is_outside_enclave(session as * const u8,
                                                mem::size_of::<TlsClient>()) {
        return -1;
    }
    rsgx_lfence();

    let session = unsafe { &mut *(session as *mut TlsClient) };
// [...]
}
```

Figure 4.13: Vulnerable Rust code: Check ② can be bypassed.

implementations with an object instance. Depending on the type of the object, the compiler will jump to the right method implementation by using the indirection via the vtable pointer. However, controlling the vtable pointer is typically equivalent to controlling the instruction pointer. The attacker can make the vtable pointer refer to attacker-controlled memory that contains a fake vtable. Subsequently, the victim code dereferences the vtable pointer to look up the method in the fake vtable and perform an indirect jump to the value from the fake vtable. As such, this attack vector has also received significant attention from the research community, developing various exploit mitigation technologies, such as multiple CFI variants [Sze+13; Tic+14].

However, the enclave performs some validation on the received pointer: it must not be null (① in Figure 4.13) and must not be outside enclave memory (② in Figure 4.13). Similar to the attack against the *WolfSSL* enclave (see Section 4.3), this could be exploited by injecting the fake *TLSSession* object into the enclave memory using some unrelated function. However, the enclave exhibits another anti-pattern that can be exploited here. Namely, the pointer validation does not account for the possible three states concerning a memory object (see Section 4.2.3). As such, the check at ② is not sufficient to sanitize a pointer.

In our PoC exploit, we chose to bypass the pointer validation using overlapping memory. We map the memory pages right before the first enclave page into the normal user space of the host application. We then place a fake *TLSSession* object at the

page boundary, such that only the last byte of the *TLSSession* object is within enclave memory. This makes the pointer validation at ② in Figure 4.13 succeed, as the object is not strictly outside enclave memory anymore. However, this gives us complete control over anything interesting contained in this object, including the vtable pointer, which now resides in host application memory. Now we simply let the vtable pointer refer to a fake vtable, where we can place the start of our code-reuse attack. Note that the last byte of the *TLSSession* object is not used during the ECALL that is exploited in our PoC, so the value of the last byte does not have any side effects on the execution of the enclave.

Even though this enclave uses a robust memory-safe language, TeeRex was able to identify a vulnerability at the interface boundary between the host and the enclave. Here the compiler-enforced memory safety guarantees of Rust do not hold anymore, as the boundary is necessarily implemented using unsafe rust or C code. This highlights the need to develop safe interface bindings for memory-safe languages, such as rust. Low-level details should not be exposed to enclave developers.

We disclosed our findings to the developers of the Rust SGX SDK, who acknowledged the issue and promptly developed a patch. Instead of using pointers as resource identifiers, they switched to using integer identifiers, which are mapped to the objects using a hashmap. As such, no pointers need to be passed over the host-to-enclave boundary. The attack surface is drastically removed, and both anti-patterns regarding pointer sanitization are removed from the enclave code.

TaLoS

The TaLoS project is an open-source enclave that was created as part of an academic project. The idea of the enclave is to provide a shielded version of a TLS library, essentially terminating the TLS-secured connection inside the enclave. However, integrating a shielded TLS library into classical software is challenging due to the tight integrating. For example, the Apache Webserver tries to directly call the API of the TLS library *libressl*. As such, the TaLoS project provides a wrapper library for the host that forwards all calls to the enclave. To make the transition to using the enclave a TLS library easier, TaLoS exposes a one to one mapping of the *libressl* API. As such, the enclave interface extensively exposes pointers that are marked as *user_check* in the corresponding EDL file. However, the enclave does not perform sufficient checks on the passed pointers. As such, TeeRex is able to identify various exploit primitives.

However, the enclave does not simply return a raw pointer as it is the case for the other TLS enclaves (see Section 4.3, and Section 4.3). Interestingly, the TaLoS enclave has a built-in shadowing mechanism: the enclave duplicates relevant data structures in both host and enclave memory [Aub+17]. Primarily, the shadowing mechanism is used to allow the host application to access fields of data structures used by the TLS library. This allows unmodified host applications to load and use the enclave. The enclave code contains manually written code to synchronize selected fields of important data structures, such as the primary `SSL` data structure. In principle, the shadowing mechanism would also allow the enclave to perform extensive validation of the received pointer arguments. However, in the version of the enclave that we analyzed, we found various issues with this shadowing mechanism due to the manual

```
1  BIO* ecall_SSL_get_rbio(SSL *out_s) {
2  ① // out_s is not checked, can be in enclave memory
3    /** Shadowing Mechanism **/
4    hashmap* m = get_ssl_hardening();
5    // returns NULL for invalid out_s
6  ② SSL* in_s = hashmapGet(m, out_s);
7    // copy arbitrary enclave memory to the NULL page
8  ③ SSL_copy_fields_to_in_struct(in_s, out_s);
9  ④ /* [...] libressl logic */
10   // copy from the NULL page to arbitrary enclave memory
11 ⑤ SSL_copy_fields_to_out_struct(in_s, out_s); // [...]
```

Figure 4.14: Relevant parts of the EDL definition and C source code of the TaLoS enclave.

approach of shadowing the data structures. The exposed API is quite comprehensive. TeeRex identified multiple exploit primitives in the enclave, including in the code that performs the data structure shadowing.

Figure 4.14 shows the relevant parts of the enclave, which implements the shadowing mechanism. Here, TeeRex discovered the *NULL* pointer dereference. However, the code contains several other issues. For example, the parameter *out_s* is not validated to lie outside of enclave memory (①). To exploit the *NULL* pointer dereference, the attacker must pass a bogus value for the *out_s* parameter. In turn, the call to *hashmapGet* will fail and return *NULL* to indicate an error. However, this return value is not further validated (②), and as a result, *in_s* becomes *NULL*. This turns the shadowing mechanism at ③ into an arbitrary read exploit primitive. The attacker can let *out_s* point to an arbitrary location inside the enclave memory. The shadowing mechanism at ③ will then copy the content from enclave memory to the page mapped at address 0.

Furthermore, the same code can also be abused by an attacker to gain an arbitrary write exploit primitive. After the logic of the *libressl* TLS library is done (④ in Figure 4.14), the synchronization code again synchronizes the enclave internal data structure with the data structure in host memory. At ⑤ the synchronization code copies the fields of *in_s* structure to the *out_s* structure. However, the attacker can make the *out_s* data structure point to enclave memory, and due to the failed hashmap lookup, *in_s* points to the page at address 0. As such, the enclave will copy arbitrary values from host memory to an arbitrary location within enclave memory. Together, this gives an attacker a powerful arbitrary read/write exploit primitive. However, the attacker has to win a race for effective exploitation, as the attacker must inspect and change the value while the code at ④ is executing. This can be achieved by using a second thread in the user space of the malicious host application. Alternatively, the attacker can reliably exploit this by interrupting the enclave using at just the right time. Previously it was shown that the timer interrupts are precise enough to essentially single-step enclave code, making it trivial to exploit such race conditions [VPS17].

Besides the arbitrary read/write exploit primitive TeeRex also discovered a control-flow hijacking exploit primitive. Namely, similar to the other TLS enclaves, TeeRex identifies a function pointer used for a callback in one of the primary data structures (the *SSL_CTX*). In contrast to other data structures, there is no shadowing of this pointer in the enclave. In fact, the enclave does not validate the pointer or the data structure at all. As such, to exploit this issue, the attacker must simply craft a fake data structure in host memory and call the vulnerable ECALL (*ecall_SSL_CTX_ctrl*).

Synaptics SynaTEE Driver

We identified several fingerprint drivers for Windows that utilize the SGX technology to securely process biometric data. The first fingerprint driver we encountered was introduced by *Synaptics* and is utilized on modern *Lenovo* laptops. The fingerprint driver delegates parts of the fingerprint-based authentication to a user-space component that utilizes a SGX enclave. In our evaluation, this is also the first closed-source enclave designed to run on the Microsoft Windows OS. Here several of the pre-processing modules of TeeRex are required to hook statically linked standard library functions for more efficient symbolic execution. Furthermore, we noticed that the enclave only utilizes two different ECALLs, a suspiciously low number given the size of the enclave. This is because the enclave multiplexes several commands over one of the two ECALLs. Using manual analysis, we noticed that while TeeRex was still able to analyze the enclave, the analysis is much faster and is able to identify more findings if we change the analysis entry point. Instead of starting at the actual ECALL entry points, we start at the entry points of the several command handlers that handle the various multiplexed ECALLs.

TeeRex discovered a control-flow hijacking primitive resulting from a NULL-pointer dereference. In the enclave code, there is a global pointer variable, which points to a data structure that is expected to be in enclave memory. However, this pointer is initialized as a NULL pointer, and the enclave features an ECALL that allows an attacker to trigger dereferencing the global pointer without initializing the global pointer first. As such, the attacker can exploit the fact that the NULL pointer refers to host memory.

However, exploiting a NULL pointer dereference on a modern Windows OS is cumbersome, as the latest Windows versions disallow mapping the page at address 0. The SGX threat model would allow an attacker to override this setting. While this cannot be easily disabled, it is possible to patch the Windows kernel to disable the check. We demonstrate the feasibility of this approach with our exploit for the Goodix enclave in Section 4.3. To achieve a pure user-space exploit for the Synaptics enclave, we chose a different route for exploitation.

In the vulnerability report, TeeRex also reports secondary findings, which are not exploitable alone, but potentially ease writing full exploits (see Section 4.1). More specifically, TeeRex identifies a limited write exploit primitive in the enclave due to an improperly sanitized data structure that contains many pointers. This primitive allows an attacker to write a single byte with a fixed value to an arbitrary address. While this primitive is not enough to achieve, e.g., full control-flow hijacking, it still allows an attacker to corrupt enclave memory and continue exploitation from there. As a first

step, we utilize this limited primitive to overwrite one byte of the NULL-initialized global pointer, such that this pointer refers to a known location in host memory. Corrupting the pointer first bypasses the need for mapping the page at address 0.

In our PoC exploit, we chain the limited write and the control-flow hijacking primitive. First, we corrupt the global pointer with the fixed value. Second, we map the memory referred to by the resulting pointer. Third, we prepare a fake data structure in the freshly mapped memory. Finally, we trigger the ECALL that dereferences the—now corrupted—global pointer and achieve a control-flow hijacking primitive.

Goodix Fingerprint Driver

The second fingerprint-reader driver we analyzed is shipped on recent laptops produced by *Dell* and is developed by *Goodix*. Similar to the fingerprint-reader driver of Synaptics, a user-space component with a SGX enclave is used. Again we performed black-box binary analysis of the enclave with TeeRex. This analysis resulted in the discovery of several limited write primitives, two of which we utilize in our PoC exploit.

The first primitive, which we denote as C_{16}, is a NULL-pointer dereference that copies a 16 bit value from the NULL pointer to an arbitrary address supplied by the attacker. As we discussed earlier for the Synaptics SynaTEE Driver PoC exploit (see Section 4.3), exploiting a NULL pointer dereference comes with significant effort on modern Windows systems. To map the page at address 0 on Windows, we need to first patch the kernel using an attached debugger. The patched kernel then simply allows mapping the address 0 in a normal user space application. Note that the *patch guard* of the Windows kernel periodically scans kernel code and attempts to detect code corruption, which further complicates exploitation.

However, since we have discovered only a 16 bit write primitive, we cannot fully overwrite pointer values to gain a stronger exploit primitive (e.g., a control-flow hijack). While a partial overwrite is often already sufficient to launch a code-reuse attack [Dur02], we opted to chain to primitives to achieve a full arbitrary write of a 64 bit value. To do so, we introduce a second limited exploit primitive F_{64}. The primitive F_{64}loads a value from a fixed address A within enclave memory but writes this value to an arbitrary attack-controlled pointer. Since the value is within enclave memory, the attacker cannot control it, and as such, this primitive is of limited use alone. However, we can use the first primitive $C_{16}$to set the value at address A. We utilize four invocations to $C_{16}$to write a full 64 bit integer. We then utilize $F_{64}$to copy this value to an arbitrary location inside enclave memory, effectively gaining an arbitrary write. We can then overwrite, e.g., a return address on the enclave stack to launch a code-reuse attack. This exploit shows that the auxiliary exploit primitives discovered by TeeRex are useful to a human analyst, who can quickly combine multiple limited exploit primitives to upgrade to a full arbitrary write exploit primitive.

4.4 Performance and Accuracy

Previously, we primarily discussed the findings of TeeRex in several enclave projects. With this, we demonstrate that TeeRex is a viable approach for enclave analysis.

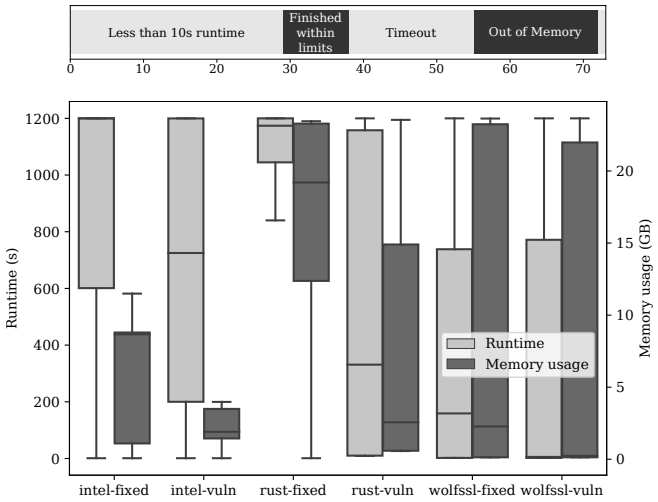


Figure 4.15: Runtime and memory usage of the benchmarked enclaves.

In this section, we discuss the efficiency and effectiveness of TeeRex as an analysis tool. We focus our analysis on the three enclaves Intel GMP Example, Rust SGX SDK's tlsclient, and WolfSSL Example Enclave since for these (1) the source code is available, and (2) a patched version is available to us. These properties allow us to analyze TeeRex analyses in more detail, as we can compare the analysis on the vulnerable and fixed enclaves. This allows us to reason about the occurrences of false alarms. First, we verified the source-level patches for the three enclaves we use for performance evaluation. Since source code is available, we can analyze the patched code in detail and determine whether a newly reported finding is a false positive. Second, we can obtain more realistic performance numbers on the patched enclaves. Unfortunately, our dataset of enclaves consists of enclaves that contain multiple pointer validation vulnerabilities. In fact the vulnerabilities are so prevalent that we cannot properly assess the performance of TeeRex. As such, we measure performance on patched enclaves to obtain measurements of TeeRex analyzing enclaves where pointer validation bugs are not as prevalent as in our current dataset of unpatched enclaves.

4.4.1 Performance and Memory Usage

When running TeeRex, we analyze each ECALL using TeeRex for a maximum of 20 min using a single CPU core and a memory limit of 24 Gbyte. The analysis was conducted on an AMD EPYC Processor with 3.7 GHz and 100 Gbyte RAM allowing us to analyze up to 4 ECALLs in parallel. TeeRex utilizes angr version 8.20.1.7 running on CPython 3.6.9 and Ubuntu 18.04.4. All the exploitable primitives that we utilized in our PoC exploits are discovered within our time window of 20 min.

For the three enclaves (Intel GMP Example, Rust SGX SDK's tlsclient, and WolfSSL Example Enclave), we analyzed the 73 ECALLs in detail. The results are depicted in Figure 4.15. Symbolic execution tends to be limited by the available memory due to state explosion. The average memory usage overall ECALLs of those enclaves is

8.8 Gbyte, with a standard deviation $\sigma = 9.8$ Gbyte). The high standard deviation can be explained by the high variability of code size and code complexity of the ECALLs. 40 % of the analyzed ECALLs finished within 10 s, Further 52 % finished within the given limits. However, 48 % exceeded the resource limits. More specifically, 23 % by time, also 23 % by memory, and 1 % by time and memory.

We chose the resource limits empirically, after performing several analysis runs. As our findings show (see Section 4.3), these time and memory constraints allow TeeRex to uncover problematic code patterns. In a more realistic setting, an analyst can invest more time and memory for specific ECALLs. Nevertheless, symbolic execution is a powerful but resource-hungry program analysis technique. There are many techniques that allow improving the efficiency of symbolic execution [Avg+14; Bal+18; Cha+12], which we did not yet integrate into our prototype.

4.4.2 Accuracy and False Alarms

We designed TeeRex with soundness in mind and typically sacrificed completeness of the analysis, i.e., using our resource-constrained analysis strategy. As such, the number of false alarms is generally relatively small. Note that we cannot provide a full analysis of false alarms and missed bug rates, as we lack a dataset with ground truth. All our findings were previously unknown zero-day vulnerabilities. Prior to TeeRex, there were no other automated vulnerability analysis tools available that are capable of identifying vulnerabilities specific to SGX enclaves. As such, we cannot compare TeeRex to prior work.

We use the following strategy to assess the accuracy of TeeRex. First, we manually confirm all findings by constructing PoC exploits. We also worked with the vendors to confirm the vulnerabilities and obtain patched versions of the enclaves. Whenever source code was available to us, we also verified, using TeeRex and manual analysis, that the vendors correctly fixed the issues that we reported. As such, we can be certain that the enclaves Intel GMP Example, Rust SGX SDK's tlsclient, and WolfSSL Example Enclave do not contain any further vulnerabilities. We can then assume that any finding of TeeRex on the patched enclave is a false alarm, giving us a limited form of ground truth. Our analysis of the three vulnerable enclaves TeeRex produced 149 findings. The large number of findings can be explained by the vulnerable code patterns that are present several times in the enclaves. By manual deduplication and with our PoC exploits, we confirm that those findings are correctly reported alarms. For each of the vulnerabilities, we selected gadgets in shallow program paths with the smallest number of constraints on the initial state. This allows us to write simpler PoC exploits.

Next, we analyzed the patched versions of the enclaves. First, we are able to confirm with TeeRex that the vulnerabilities we originally exploited are no longer present in the enclave code. However, the analysis of TeeRex still identified 56 major findings. Analysis of those findings reveals a source of false alarms in TeeRex: global memory is treated as unconstrained symbolic memory (see implementation details Section 4.1).

In TeeRex, we decided to mark all global enclave memory as unconstrained symbolic memory. This allows TeeRex to analyze program paths that would require a sequence of ECALLs without executing a sequence of ECALLs. We observed a common pattern

in enclave code where certain ECALLs first require a preceding call to another initializer ECALL. For example, we identified this pattern in the Intel GMP Example, which utilizes a special ECALL that initializes a function pointer in a global function pointer. By marking the global enclave memory as unconstrained values, TeeRex reports a controlled jump. This is because TeeRex does not identify any further constraints during the execution of the ECALLs. As such, TeeRex believes that the function pointer is not checked before its use. However, in reality, the enclave offers only a limited—and safe—number of ways to set the jump target. As such, this finding is not exploitable on its own. We identified similar false positives in some of the other patched enclaves. The Rust SGX SDK's tlsclient and WolfSSL Example Enclave enclaves show false alarms due to similar issues. However, it is not clear how to handle such findings because the same finding can become useful as part of a longer exploit chain. For example, it could be used to turn an arbitrary write primitive into a control-flow hijacking primitive. As such, it would be detrimental to completely filter such findings. In future work, it would be beneficial to extend TeeRex with a post-processing analysis that ranks findings according to severity based on further analysis (e.g., using the pointer tracking component).

4.5 Discussion and Conclusion

SGX is a promising upcoming security technology that can be used to strongly isolate sensitive code and data into enclaves. SGX greatly reduces the TCB. However, it also creates a new attack surface: the host-to-enclave boundary becomes highly critical and dangerous. By design, the enclave processes and operates on input originating from untrusted memory space. Implementation errors in the code handling the boundary are easy to introduce and quite fatal. Based on our real-world findings with TeeRex, we describe several vulnerability patterns that are easy to introduce in code written for the Intel SGX SDK. Our analysis shows that these patterns are especially common when porting legacy code to SGX.

In this chapter, we present TeeRex, the first automated vulnerability analysis tool explicitly tailored to SGX enclave binaries. We show that symbolic execution is a fruitful and feasible approach to automatically vet enclave code for bugs on the host/enclave interface. TeeRex uses symbolic execution to precisely reason about program paths in the enclave interface and discover memory corruption vulnerabilities. Going even further, TeeRex reports memory corruption vulnerabilities as exploit primitives, such as arbitrary writes or control-flow hijacking. This allows an analyst to quickly construct PoC exploits to determine the severity of the findings.

Limitations While symbolic execution is a powerful program analysis technique, it has several drawbacks that TeeRex inherits. First, we attempt to mitigate state explosion by implementing an ECALL centric analysis in TeeRex. This means we can analyze available ECALLs in parallel, exploiting multi-core systems for analysis. However, we noticed several cases where this ECALL centric analysis is not enough to mitigate state explosion. For example, we identified one enclave that essentially multiplexes several ECALLs over one ECALL. Running the default analysis of TeeRex would result in a path explosion problem as TeeRex would not detect this multiplexing, leading to a large number of analysis states. In the case of this enclave, we were able to circumvent this issue by starting analysis at the handlers for the multiplexed ECALLs. However, this approach requires manual analysis and is not applicable to all enclaves.

Furthermore, the symbolic environment that TeeRex uses to emulate the target enclave does not precisely replicate the real enclave environment, which causes TeeRex to miss certain bug classes. For example, TeeRex replaces the allocator of the enclave with a simplified simulated allocator that is provided by the Angr framework if it can identify the enclave's allocator (e.g., using symbols). The emulated allocator always returns a new memory address and never returns a memory address that is already in use. This simplifies symbolic execution and avoids costly symbolic execution of the memory allocator. However, the downside of this approach is that temporal memory errors cannot be identified anymore. For example, because the emulated allocator does not reuse memory, TeeRex will never encounter a use-after-free. However, in large C++ code bases, temporal vulnerabilities make up for the bulk of the vulnerabilities [Tho19; Zer]. Detecting temporal vulnerabilities would require extending TeeRex with more precise analysis and modeling of the enclave heap during analysis [Eck+18; Gri+22]. Alternatively, TeeRex could be used as the basis for a hybrid symbolic execution and

fuzzing approach, where it is much easier to detect temporal vulnerabilities such as use-after-free issues [Clo+22; Sho+16; Ste+16].

Finally, TeeRex also does not model the full set of interactions that are possible. For example, TeeRex only approximates ECALL sequences by marking global memory as unconstrained symbolic. This allows TeeRex to explore paths within an ECALL that would normally only be reachable with multiple ECALLs. However, this is also the single source of false alarms that we noticed during our evaluation of TeeRex (see Section 4.4). Naturally, this approximation could be avoided by chaining multiple ECALLs during analysis. However, this introduces a simple combinatorial path explosion into the symbolic analysis process: after every ECALL, the attacker can call every other ECALL. Practically speaking, chaining ECALLs would also entail modeling the OCALL mechanism. Currently, TeeRex stops execution when it encounters an enclave exit, whether the enclave exit is caused by an OCALL or because of a regular enclave return. If the enclave has support, one would also need to consider reentrant ECALLs that can happen during an ongoing OCALL. This combinatorial path explosion is common to all secure execution environments that expose a library-like interface. We discuss tackling analysis of full call chains, including reentrancy, in Chapter 5 using Ethereum smart contracts as our targets.

Conclusion TeeRex uses symbolic execution to analyze the host/enclave interface. Using TeeRex, we identified several vulnerabilities in public enclaves, including two fingerprint drivers developed by Synaptics and by Goodix, three TLS libraries, and a project published by Intel. We developed PoC exploits based on the exploit primitives discovered by TeeRex. The report produced by TeeRex is detailed enough to quickly construct such exploits. While TeeRex inherits the limitations of symbolic execution, primarily dealing with path explosion, the results we present in this chapter show that our analysis approach can be used to identify real vulnerabilities in enclave code. Furthermore, it allowed us to perform the systematic analysis of enclave code and discover several vulnerability patterns common to multiple enclaves. Enclave developers or security analysts can now avoid these vulnerability patterns before deploying enclave code. Furthermore, our work on TeeRex inspired further investigation into automatic analysis of SGX enclave code using fuzzing techniques that scale better to larger enclaves [Clo+22].

CHAPTER 5

FUZZING OF SMART CONTRACTS

Ethereum is the most prominent blockchain platform that supports *smart contracts*, i.e., programs that are stored and run as part of the blockchain protocol. Currently, smart contracts form the backbone of the emerging decentralized finance (DeFi) industry. We discuss the Ethereum execution environment in detail in Section 3.2.

Due to its widespread popularity, the security of Ethereum, particularly it's smart contract layer, has received considerable attention from the research community and industry. Despite their popularity, current smart contract development practices and tooling still lag significantly behind the state-of-the-art in software security. This became evident after a series of high-profile attacks that targeted a number of popular smart contracts in Ethereum, such as "the DAO" attack [Jen16], among others [PL21; Tor+21b; Zho+20].

Analyzing smart contract code is highly challenging due to the stateful nature of smart contract code. Each smart contract exposes a certain set of functions that can be used to interact with the smart contract in the form of atomic transactions. Most of these interactions change the internal state of the contract, and software faults typically only manifest when the smart contract is in a particular state. Automatically testing and analyzing stateful software is generally a highly challenging problem (see Section 2.2). This is further exacerbated by the fact that many smart contracts call into other smart contracts, which in turn can reenter the originally called smart contract again. This is a form of deterministic concurrency attack (see Section 2.1.2), which has as previously led to many high-profile vulnerabilities called as *reentrancy attacks* [Jen16; Rod+19].

Previously, in Chapter 4, we explored symbolic execution as a technique to automatically identify vulnerabilities in stateful software that exposes an API. However, we also discussed several problems with symbolic execution, most importantly, the scalability issues when analyzing sequences of API calls. This chapter presents a state-of-the-art fuzzing framework for Ethereum smart contract called Extremely Fast Contract Fuzzer (EF↯CF). We show that in contrast to symbolic execution and prior Ethereum fuzzers, EF↯CF scales better to long and complex sequences of API calls. EF↯CF faithfully handles complex interactions, such as reentrant executions and cross-contract calls.

Contributions To summarize, the main contributions presented in this chapter are:

- We devise a novel transpilation approach to accelerate the fuzzing of bytecode programs, such as the EVM. Our approach removes the interpreter by directly translating bytecode into equivalent C++ code and, finally, native code (Section 5.2). This allows us to reuse high-speed native fuzzing components for coverage instrumentation.
- We design the first fuzzer that can efficiently and accurately generate complex reentrancy attacks (Section 5.3). In contrast to prior analysis tools, we do not over-approximate reentrancy attacks but generate and execute them directly. We do this by letting the fuzzer choose the behavior of several simulated attacker smart contracts. The fuzzing process explores the different behaviors of the attacker's smart contracts, guided by code coverage of the target smart contract. The result can be translated to a set of attack contracts in the Solidity programming language.
- We thoroughly evaluate the performance of EF↯CF against a large number of state-of-the-art analysis tools. We show that our approach scales better to complex contracts (Section 5.4) without hampering its effectiveness (accuracy and coverage) on other non-complex contracts (Section 5.5). Furthermore, we show that EF↯CF is capable of accurately identifying even complex real-world compositional reentrancy issues, such as the *Uniswap/IMBTC* incident (Section 5.5.2).

The topics discussed in this chapter have been previously presented in the following publication:
"EF/CF: A High Performance Fuzzer for Ethereum Smart Contracts". ***IEEE European Symposium on Security and Privacy (EuroS&P)*, 2023**. Michael Rodler, David Paaßen, Wenting Li, Lukas Bernhard, Thorsten Holz, Ghassan O. Karame, and Lucas Davi

5.1 Challenges of Automated Smart Contract Analysis

When fuzzing smart contracts, the goal is to identify a sequence of transactions that exposes a fault in the smart contract. The final transaction of the sequence triggers a fault, while the preceding transactions set up the state of the contract such that the fault can be triggered. As such, testing Ethereum smart contracts represents a highly challenging problem: a variant of testing stateful software [Ait02; CH00; KLT01; Thu+11; Xie+05]. In contrast to static analysis methods, generating complete transaction sequences has the advantage that it features a very low rate of false alarms. Furthermore, the result is easy to analyze: a developer or security analyst can simply replay and debug the transaction sequence to determine the root cause and assess whether the bug can be triggered in practice. However, determining such a transaction sequence is challenging since the search space is extremely large. There are two dimensions that must be explored in parallel to reach high code coverage: (1) the input to individual transactions and (2) the ordering of the transactions. To

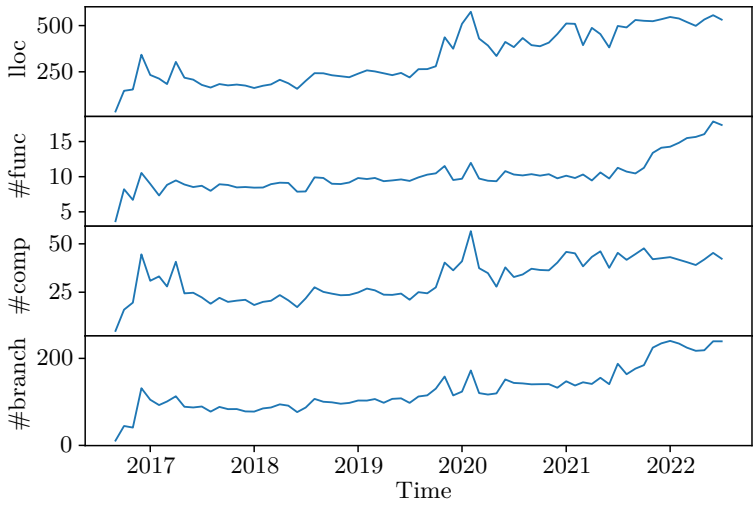


Figure 5.1: Increasing trend in smart contract complexity over time. We measure the complexity of all unique contracts with verified source code that appear in Ethereum until April 27, 2022, and report the average metrics across all contracts deployed.

efficiently cover this large search space, we can exploit the fact that many of the possible transaction sequences are redundant or simply pointless since they only exercise the same error-handling paths repeatedly. *Coverage-guided fuzzing* can efficiently search the input space of a given contract for inputs that trigger distinct code coverage, and was popularized by the success of the American Fuzzy Lop (AFL) fuzzer [Zal] and many follow-up works [Fio+20; Man+21].

In this context, test case throughput emerges as an important design aspect for an effective fuzzer. Intuitively, the greater the number of test cases generated/executed, the greater the likelihood that a fault will be triggered within a given time budget during fuzz testing, an inherently probabilistic process. Most fuzz testing approaches for Ethereum smart contracts develop new fuzzers from scratch [Gri+20; He+19; Ngu+20; Tor+21a]. In doing so, existing smart contract fuzzers neglect years of engineering effort that has been invested by the community into the development of fast fuzzing strategies of native code (typically C or C++). For example, ILF performs at a rate of 148 transactions per second [He+19], while native code fuzzing with far more complex code regularly achieves 10 000 or more test case executions per second [Xu+17]. Due to their low throughput, current smart contract fuzzers (1) cannot scale well when testing complex contracts and (2) cannot accurately model complex interactions with attacker-controlled smart contracts. Next, we discuss these limitations in greater detail.

Increasing Complexity of Smart Contracts Smart contracts are being used to implement increasingly complex business processes. As a consequence, also the complexity of the smart contracts according to various common complexity metrics is also increasing over time. Our measurements of various complexity metrics for smart

contracts as they evolved over time are depicted in Figure 5.1. More specifically, we analyzed the source code of 120 556 unique contracts, which were deployed to the main Ethereum blockchain until July 22, 2022. Note that this also includes all the contracts from the *smartbugs-wild* dataset [Dur+20]. For each contract, we measure the number of logical lines of code[1], the number of state-changing public functions[2], the number of comparison operators, and the number of branches in the control flow of the smart contract bytecode[3]. Our analysis confirms the anecdotal increase in smart contract complexity across all complexity metrics we analyzed. The complexity problem is further increased by an increasing trend of smart contracts being composed into systems of multiple interacting smart contracts. While we do not know of any measurements of contract coupling of modern smart contracts, multiple real-world *compositional* security vulnerabilities act as anecdotal evidence of this problem [Cec+21].

Multiple consecutive transactions are required to exercise all code paths while testing the ever-increasing complex smart contracts. To reach all code paths, one needs to explore the internal states of a smart contract. However, most prior studies are only limited to rather short transaction sequences of length 3 [FAH20; KR18a; Nik+18] or do not assess the ability to work with longer transaction sequences [Gri+20; He+19; Tor+21a; WC20]. Typically, more complex contracts also require longer transaction sequences to cover different states of the contract during testing. To assess the ability of prior analysis tools to cope with longer consecutive transaction sequences, we conducted an experiment with a set of benchmark contracts with artificial bugs. Our experiment, detailed in Section 5.4, shows that existing analysis tools are not sufficient to analyze more complex contracts. In particular, current fuzzing-based analysis tools [Gri+20; Tor+21a] were unable to identify the bugs that require a specifically ordered sequence of six or more transactions. While symbolic execution tools [KR18a; Mos+19; Nik+18] are capable of producing such sequences even up to ten transactions, they fail to identify faults that require accumulation of internal state over multiple transactions. Apart from this, none of the analysis tools we tested can identify all bugs within a generous time budget of 48 hours.

Support for Complex Smart Contract Interactions Another challenge that we need to tackle is the frequent interaction of smart contracts with each other. To precisely model such an interaction, whenever a smart contract calls another (potentially untrusted) smart contract, we must assume that the target smart contract can be reentered at any function. Such so-called reentrancy attacks have had devastating consequences in the past and even required a hard fork of the Ethereum blockchain. To faithfully emulate the attacker's capabilities with respect to reentrancy, we must simulate the following scenarios: At each call to an untrusted, potentially attacker-controlled contract, the target smart contract can be reentered (1) at the same call depth multiple times, (2) at multiple functions, and (3) by a call originating from a different smart contract. As such, accurately modeling an attacker capable of performing these variants and combinations of reentrant transactions is highly challenging. Current

[1] lloc is counted with github.com/XAMPPRocky/tokei.
[2] #func is counted as the number of ABIs with non-constant attribute.
[3] counted using the EtherSolve [Con+21] static analyzer

analysis tools mostly refrain from modeling arbitrary reentrant transactions due to state explosion [Kal+18]; instead, most tools utilize over-approximative detectors for reentrancy bugs (e.g., no state updates after calls [FGG19; Tor+21a; Tsa+18]).

To better illustrate the challenge, consider the example in Figure 5.2, which depicts a token-like contract with standard transfer and allowance mechanisms that is vulnerable to a reentrancy attack. If using the *checks-effects-interactions* code pattern [SolCEI] is not possible, the second best alternative to prevent reentrancy attacks is to use locking mechanisms [SolRe]. However, many analysis tools, such as Slither [FGG19], Securify [Tsa+18], and Confuzzius [Tor+21a], do not handle locking mechanisms appropriately and simply report a potential reentrancy issue in the *withdrawBalance* function, both due to the locking mechanism and the balance update. In the example in Figure 5.2, the modifier *withdrawAllowed* prevents an attacker from calling the *withdrawBalance* function in a reentrant manner. This gives the developer a false sense of security, thinking that the *userBalances* variables are protected by the locking mechanism and as such the contract must be secured against reentrancy attacks. However, this only assumes that the attack follows the call chain $A_1 \rightarrow B \rightarrow A_1 \rightarrow B$. Since the attacker A_1 has arbitrary control, they can transfer control to a different colluding smart contract A_2, which allows executing the call chain $A_1 \rightarrow B \rightarrow A_1 \rightarrow A_2 \rightarrow B$. As such, the second attacker contract A_2 can call into the *transferFrom* function to move away the balance of A_1 before the call to *withdrawBalance* finishes and resets the balance. With this attack, it is possible to bypass the reentrancy locking mechanism and withdraw twice the balance that should be available to the attacker.

```
contract Bank {
  mapping(address => uint256) balance;
  mapping(address => bool) disableWithdraw;
  mapping(address => mapping(address => uint256)) allow;

  modifier withdrawAllowed { // reentrancy locking
    require(disableWithdraw[msg.sender] == false); _; }

  function addAllowance(address other, uint256 amnt) public
  { allow[msg.sender][other] += amnt; }

  function transferFrom(address from, uint256 amnt) withdrawAllowed public {
    require(balance[from] >= amnt);
    require(allow[from][msg.sender] >= amnt);
    balance[from] -= amnt;
    allow[from][msg.sender] -= amnt;
    balance[msg.sender] += amnt; }

  function withdrawBalance() withdrawAllowed public {
    // set lock
    disableWithdraw[msg.sender] = true;
    // reentrant calls possible here
    msg.sender.call{value: balance[msg.sender]}("");
    // release lock
    disableWithdraw[msg.sender] = false;
    balance[msg.sender] = 0; }
  /* ... */
}
```

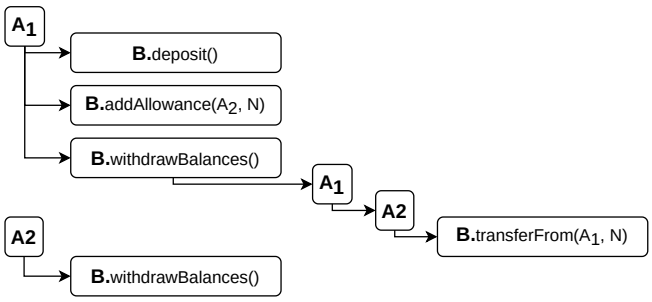


Figure 5.2: A contract, B, with bypassable reentrancy-locking using multiple colluding attacker-contracts (A_1 and A_2). The reentrant transaction sequence depicted below the code listings exploits the shown contract, bypassing the reentrancy-locking.

5.2 Design of EF⚡CF

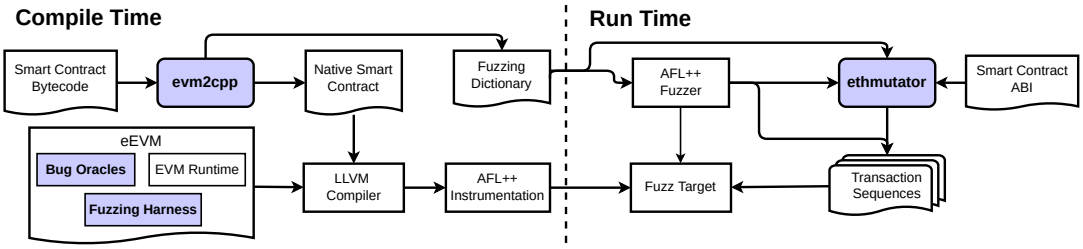


Figure 5.3: Architecture of EF⚡CF. Blue boxes are newly developed or modified components.

The design of EF⚡CF is driven by two major features: optimizing test case throughput and accurately modeling complex interactions with smart contracts. To achieve the former, EF⚡CF uses two explicit phases, a compile and a run time phase (see Figure 5.3). At compile time, the EVM bytecode of the smart contract is translated to C++ code with our newly developed *evm2cpp* compiler and paired with a fuzzing-optimized EVM runtime, facilitating fast smart contract execution. To accurately model interactions with smart contracts, we devise an approach to allow the fuzzer to mutate the behavior of multiple simulated attacker-controlled smart contracts. Each generated test case specifies a sequence of transactions, which are executed by the fuzzing harness. However, in contrast to prior work [FAH20; Gri+20; KR18a; Nik+18; Tor+21a], this transaction sequence also specifies the behavior of callbacks to attacker accounts, including return values and further reentrant transactions. To detect bugs, EF⚡CF features detectors that are directly built into the EVM runtime and the fuzzing harness. Here EF⚡CF supports a commonly featured Ether-based bug oracle that attempts to gain Ether, but also custom bug oracles that are specified by a developer in Solidity code. In what follows, we discuss and explain our design choices in more detail.

5.2.1 Modelling Blockchain Interaction

To faithfully model complex (possibly adversarial) interactions on the blockchain, we define an input format for Ethereum transaction sequences that supports return data and reentrant transactions. EF⚡CF runs the smart contract in a custom blockchain environment, which contains several attacker-controlled accounts. A user of EF⚡CF can also supply a custom initial blockchain state, e.g., to fuzz smart contracts which rely on other smart contracts or expect to be deployed at a certain address. Given that smart contracts can depend on various environmental data that is retrieved from the Ethereum blockchain, EF⚡CF allows the fuzzer to choose and mutate these values at will. For example, the fuzzer chooses the block number and timestamp at the beginning of the transaction sequence and is allowed to advance both at every transaction. This enables us to handle smart contracts that expect a certain timespan to pass between two consecutive transactions. Furthermore, the fuzzer can increase the initial Ether balance of the target contract to simulate prior Ether investment into the contract.

The blockchain state is reset before every executed test case, which ensures that each generated transaction sequence can be deterministically executed. This is necessary to obtain reliable coverage measurements and eases root-cause analysis since the developer can reliably replay a transaction sequence. In many cases, the transaction sequence can be directly utilized as an end-to-end exploit against the deployed version of the contract.

Every test case in EF↯CF consists of a header specifying the initial environment followed by a sequence of transactions. Similar to regular Ethereum transactions, each transaction consists of a sender, a receiver, a call value (i.e., transferred Ether), and associated input data. However, for performance reasons, we restrict the senders and receivers to a small set of accounts that are fixed when the fuzzer is launched. In a typical single-contract fuzzing setup, the set of receivers will include only the target smart contract. EF↯CF simulates the behavior of arbitrary smart contracts at the attacker-controlled accounts. Each transaction has additional associated data beyond what a regular Ethereum transaction requires. This includes fields that specify what to do if the target smart contract calls back to an attacker-controlled address. Each transaction can have multiple associated *return-headers*, which specify (1) whether the call succeeded, (2) what data to return, (3) and how many reentrant calls can be performed. The fuzzer is then free to choose arbitrary values for any of these parameters. This allows the fuzzing process to explore a large variety of behaviors of attacker-controlled smart contracts.

Reentrant Transactions As described in Section 5.1, it is also important that we model complex interactions with the target smart contract, including reentrant transactions. Prior dynamic analysis tools [JLC18; KR18a; Tor+21a] focused on generating lists of transactions that trigger an exploit. However, to also model reentrant transactions, a simple list data structure is not sufficient: every top-level transaction can have multiple associated reentrant transactions of which every transaction can have again associated reentrant transactions (see Figure 5.2 for an example). As such, the execution of a transaction spans a tree of calls to other contracts. To let the fuzzer simulate the behavior of a reentrancy-capable attacker, we need to model this call-tree in the input format for the fuzzer. Mutating the tree structure allows the fuzzer to explore various shapes of the call-tree: reentering the same function repeatedly, reentering the same function only once, reentering the same contract in a different function, or reentering the same contract multiple times at the same call-depth.

However, in practice, not all shapes of the call-tree are possible. Some functions of a contract allow for callbacks to the attacker, and therefore further reentrant transactions, while others do not. In general, it is not possible to compute the shape of the call-tree in advance. Whether an external call to an attacker is performed by the target smart contract generally depends on the input and as such, cannot be determined before executing the transaction.

Therefore, EF↯CF's fuzzing components operate on a list of transactions. However, the fuzzing harness of EF↯CF treats this list as a queue and dynamically builds a tree of transactions, i.e., when an external call is encountered, it will reenter with the next transaction in the queue. To mutate this ad-hoc tree structure, the fuzzer can mutate

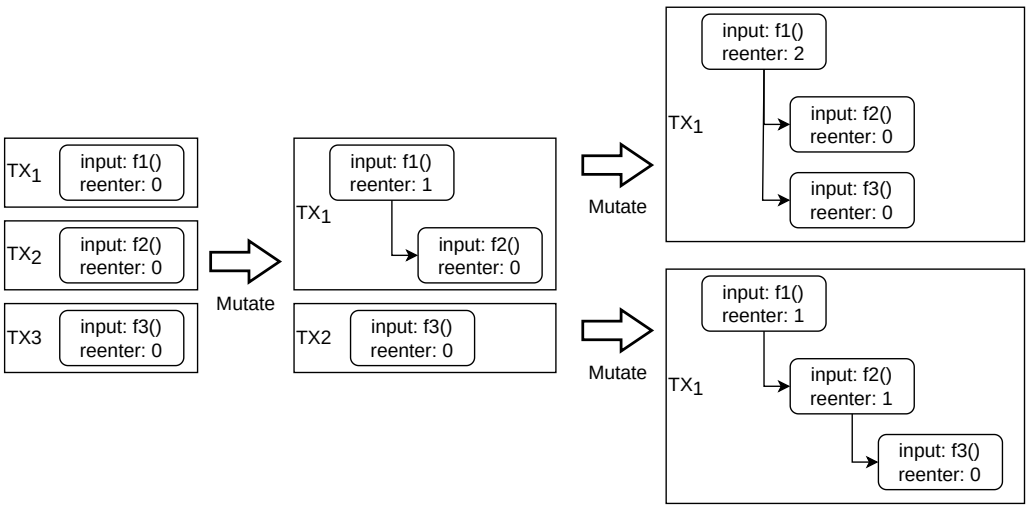
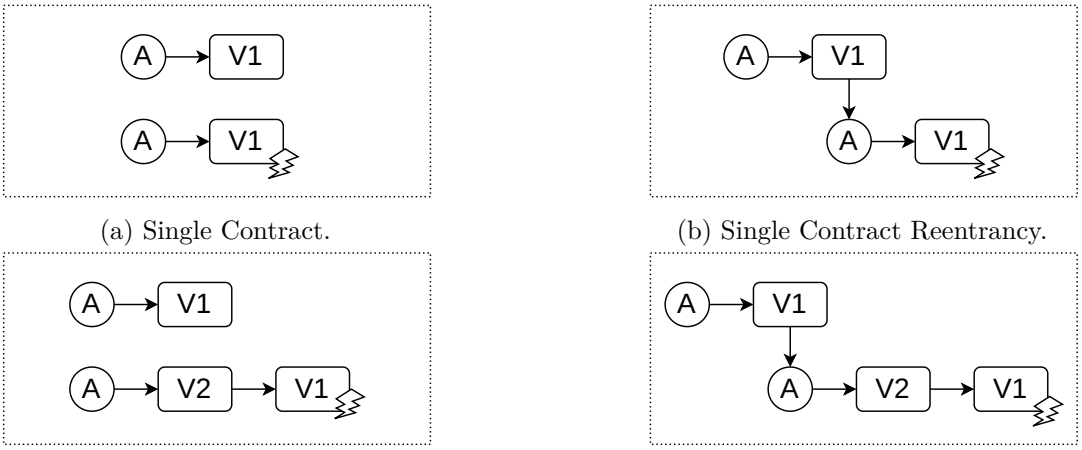


Figure 5.4: Mutating a flat transaction sequence to obtain reentrant transaction sequences with different shapes. The fuzzer modifies the *reenter* flag associated with the various transactions.

a single field in the return header that specifies how many reentrant transactions can be executed when an external call is encountered. The fuzzing harness ignores this field if the transaction does not trigger an external call. For example, Figure 5.4 shows the mutation EF⚡CF performs to create reentrant transaction sequences. We start with a flat sequence of three transactions that target the functions *f1*, *f2*, and *f3*. Both transactions have the *reenter* flag set to 0, if the function *f1* attempts to call back into the attacker, the call will fail. The fuzzer then probabilistically mutates the *reenter* flag and sets it to 1 in the first transaction. Now the same callback will not fail anymore but instead make the second transaction a reentrant transaction. The third transaction remains a top-level transaction. With this simple mutation, EF⚡CF has now generated a cross-function reentrancy attack. By changing the *reenter* flag in different transactions EF⚡CF can generate different tree shapes. Figure 5.4 shows two further mutations on this first mutated test case that results in two distinctly shaped call trees.

Compositional Security Modern smart contracts are increasingly coupled with other smart contracts. For example, token contracts are often tied to exchange contracts, where the token can be traded for other tokens or Ether. Recently, several attacks have been reported that were only possible due to composition of multiple smart contracts that were independently developed [REVST; CREAM; Cec+21; Tor+21b]. For example, the *Uniswap* reentrancy attack was only possible, because the first version of the Uniswap contract was combined with a new type of token contract that would perform a callback to the attacker. The Uniswap contract did not expect reentrancy to be possible on an external call and is indeed safe when paired with most token contracts.

Compositional security is often associated with reentrancy, because the most prominent examples of compositional security issues were also reentrancy attacks. However, compositional attacks are not necessarily reentrancy attacks. Figure 5.5 shows four



(a) Single Contract.

(b) Single Contract Reentrancy.

(c) Contract Composition.

(d) Compositional Reentrancy.

Figure 5.5: Different settings illustrating the difference and similarities between reentrancy and compositional attacks.

different attack settings, where the last (sub-)transaction triggers a bug in contract *V1*. Figure 5.5a depicts a flat sequence of transactions: two subsequent calls to contract *V1*. Figure 5.5b shows a simple reentrancy attack, *V1* call back into the attacker *A*, which performs a reentrant call to *V1*. Figure 5.5c depicts a compositional attack, where a composition of two contracts is vulnerable. The DoS attack against the *Parity Multisig Wallet* is an example of such an attack [Tec17a]. The bug can be triggered by setting up a vulnerable state and then forcing *V2* to call *V1*. Figure 5.5d depicts a compositional reentrancy attack, where the call from *V2* to *V1* is a reentrant call. The *Uniswap* attack is an example of such a compositional reentrancy bug [Cec+21].

EF/CF's design also allows for testing compositions of smart contracts. First, EF/CF is designed such that it can import parts of the blockchain state to set up a realistic environment for the target smart contracts. For example, developers could test the security of their deployed contract compositions by setting up their initial state on a local test chain and then running EF/CF to detect potential issues. Second, EF/CF supports selecting the receiver of a transaction, including reentrant transactions. As such, a developer or analyst just needs to configure the set of contracts to be analyzed by EF/CF in composition.

5.2.2 Optimizing Test Case Throughput

Most state-of-the-art analysis tools [FAH20; Gri+20; KR18a; Mos+19; Nik+18; Tor+21a] develop or utilize custom EVM implementations that offer the necessary introspection and extension possibilities necessary to perform dynamic smart contract analysis. Similarly, as part of our high throughput fuzzing framework, we develop an execution environment for smart contracts that is optimized for the fuzzing use case. Here, test case throughput, i.e., both fast mutation/generation and fast execution, emerge as one of the most important properties of a fuzzer to achieve good results in practice.

Translating EVM to C++ Most widely used Ethereum clients implement an interpreter to execute EVM smart contracts. Typically, Ethereum nodes execute a large number of different smart contracts throughout their operation; in this case, it suffices to rely on an interpreter. However, in a fuzzing setting, the same smart contract is repeatedly executed. As a consequence, the overhead of the interpreter adds up over time and causes significant overhead over longer fuzzing periods. To reduce the interpreter overhead, we develop a custom translation layer from EVM bytecode to C++ and pair this with a customized EVM runtime optimized for fuzzing. EF↯CF's compiler *evm2cpp* reduces this overhead by translating the bytecode to optimized C++ code. Translating bytecode targeting a virtual machine to C++ [Sch+16; il2cpp] or directly to native code [GRAAL; Wim+19] has been previously applied in various academic and industry settings to speed up execution time. In EF↯CF, we apply this technique for the first time to EVM bytecode.

Note that a typical interpreter is implemented as a loop that fetches the next opcode and dispatches to the corresponding opcode handler. In *evm2cpp*, we eliminate the overhead incurred by the interpreter loop by mapping the EVM basic blocks directly to lexical C++ blocks. Within a basic block, the interpreter's opcode handlers are invoked sequentially. This avoids having to explicitly dispatch to each opcode handler for each instruction. Additionally, we optimize each basic block by performing constant folding and eliminating EVM stack operations to reduce memory accesses. The control-flow transfers between EVM basic blocks are mapped to C++ *goto* statements. This leads to a native version of the smart contract that can be further optimized by a standard compiler and fuzzing toolchain.

Optimizing the EVM runtime The smart contract that is translated to C++ still requires an EVM runtime that implements the opcode handlers and interaction with the blockchain. As such, we pair the C++ code generated by *evm2cpp* with an EVM runtime, which we adapted and optimized from the *eEVM* project [eEVM]. We chose the *eEVM* project because of its relatively simple codebase that can be easily extended and adapted for fuzzing. This includes omitting or simplifying several features required by a full EVM implementation operating as part of an Ethereum node. For example, our EVM runtime does not feature instruction-accurate gas tracking. We do not need the gas mechanism to limit execution time since it is limited per test case by the fuzzer. Detecting the majority of vulnerabilities, such as access control or reentrancy, does not require instruction-accurate gas tracking. However, checking the gas budget is necessary to accurately execute external calls.

Furthermore, we stop test case execution at the first failing transaction. Since the failing transaction would be rolled-back, this has no effect on subsequent transaction executions. Instead of performing roll-backs after every failing transaction, we stop the execution of the current test case, reset the state of the Ethereum blockchain to its initial state, and let the fuzzer generate a new test case. This approach nudges the fuzzer to generate transaction sequences that only contain succeeding transactions. The only exception is the last transaction in the sequence, which is allowed to explore error-handling paths. Furthermore, this approach increases the effectiveness of test case splicing: when EF↯CF combines two previously generated test cases into one, it will very likely generate a new test case containing only succeeding transactions.

Optimizing the Mutations At run time, the base fuzzer launches the target smart contract with seed inputs, i.e., a seed with a single transaction without any input. The fuzzing process is driven by a greybox fuzzing approach which involves mutating inputs, executing the fuzz target, and measuring the code coverage to find new interesting transaction sequences. Note that the base fuzzer is unaware of the structure that is inherent to the test cases, i.e., the structure of the transaction sequence and the structure of the individual transaction inputs. As such, the base fuzzer's mutation strategies are not efficient in mutating the structure of the input. Hence, we augment the base fuzzer with a carefully engineered and optimized test case generator and mutator that performs structure-aware mutations and generations. Our custom mutator is called *ethmutator* and performs both (1) mutations on the transaction inputs according to the smart contract's ABI and (2) structural mutations on the transaction sequence.

5.2.3 Bug Oracles

Prior work on smart contract fuzzing attempted to address two orthogonal problems at the same time: triggering bugs and detecting bugs [Cho+21; He+19; JLC18; Ngu+20; Tor+21a]. Bug oracles are dynamic analyses that signal the fuzzer that a fault was triggered. In this paper, we focus primarily on the aspect of triggering faults by developing a high throughput fuzzer to identify the right input to trigger a fault. We opt to primarily use a simple—yet powerful—bug oracle: Ether gains. This is in line with prior work on exploit generation for smart contracts [FAH20; KR18a; Nik+18]. However, we also support custom bug oracles defined in Solidity code by the smart contract developer to cover contract-specific bug classes.

EF↯CF defines an attack to satisfy one of the following conditions: (a) the attacker is able to trigger a *selfdestruct* to an arbitrary address, thereby allowing the attacker to drain the funds of the contract and create a *Denial-of-Service* scenario, (b) the attacker can redirect the control flow to an arbitrary address (using the `DELEGATECALL` instruction), which would give the attacker control over the target's Ether, and (c) the attacker is able to gain Ether by interacting with the contract, i.e., the sum of the balances of the attacker-controlled contracts must exceed the initial sum of balances of these contracts. In contrast to other bug oracles, this approach avoids a high number of false alarms by design. For example, accurately detecting integer overflows [FSS18] or reentrancy [Rod+19] without high-level type information is highly challenging. However, it is comparably straightforward to detect if a fuzzer generates a transaction sequence exploiting an integer overflow or reentrancy to gain Ether. Interestingly, we found that this type of bug oracle is also *more accurate* than bug oracles implemented in other analysis tools. For example, the contract depicted in Figure 5.6 is not identified as vulnerable by neither of two state-of-the-art hybrid fuzzers, Confuzzius [Tor+21a] and Smartian [Cho+21]. On the other hand, EF↯CF's Ether-gains bug oracle turns out to cover more cases such as this example.

However, an Ether-based bug oracle often does not capture the semantics of some smart contract applications, such as token contracts. As such, we also implemented support for custom invariant checking or assertion checking. This approach is also used

```
contract SimpleEtherDrainOther {
 function withdraw(address payable to) public {
   require(msg.sender != to);
   to.transfer(address(this).balance);
 }
 function deposit() public payable {} }
```

Figure 5.6: Contract that is not considered as vulnerable by the *leaking Ether* bug oracles of Confuzzius [Tor+21a] and Smartian [Cho+21], but detected by EF⚡CF's Ether gain bug oracle.

in commercially used fuzzers [Gri+20; Conc]. This allows smart contract developers to manually specify invariants that the fuzzer tries to invalidate.

5.3 Implementation Details of EF⚡CF

We now present the description of our prototype implementation of EF⚡CF.

5.3.1 EVM to C++ Translation

The *evm2cpp* component is a custom compiler that translates EVM bytecode to C++ function calls, implemented in roughly 2500 lines of Rust code. First, we perform a linear pass over the EVM bytecode to build the set of all basic blocks. We use the fact that in EVM bytecode, all jump destinations are marked with `JUMPDEST` instructions. As such, we can efficiently identify the boundaries of a basic block by looking for branching instructions and jump destination markers. However, we do not construct a full CFG, as this would require complex and error-prone analysis since all jumps in the EVM are indirect jumps. Instead, to keep the required analysis feasible, we perform local analysis and optimization at the EVM basic block level. We emulate basic blocks in isolation using abstract values as placeholders for non-constants to perform data-flow analysis and constant propagation concerning the EVM stack. Note that, in contrast to abstract interpretation, we stop emulation at the end of a basic block and, as such, do not need to handle control-flow instructions.

The code generation procedure translates each EVM basic block to a C++ lexical block. If we can infer the jump target at the end of a basic block with our constant propagation, we directly emit a *goto* statement to the target C++ lexical block. Otherwise, we have to fall back to using a large jump table via the computed goto feature to efficiently dispatch to the next basic block at runtime. In both cases, the C++ gotos are translated into a jump instruction by the C++ compiler, which is then instrumented by the coverage-instrumentation pass of the fuzzer. Effectively, this also provides edge-coverage information without the need for explicit instrumentation of the EVM code, which further improves the performance.

Within a basic block, each opcode is translated to a call to the respective opcode handler function in the EVM runtime. However, we do not use the EVM stack within basic blocks. Instead, we translate the stack-based EVM opcodes into a lightweight

```
JUMPDEST
CALLVALUE
DUP1
ISZERO
PUSH2 0x66
JUMPI
```

```
pc_5a : { /* JUMPDEST */
  /* CALLVALUE */
  const uint256_t v_1_0 = callvalue_v();
  /* DUP1; ISZERO */
  const uint256_t v_3_0 = iszero_v(v_1_0);
  /* PUSH2 0x66; JUMPI */
  if (v_3_0) {
    ctxt->s.push(v_1_0);
    goto pc_66;
  }
  ctxt->s.push(v_1_0);
  goto pc_62; // fallthrough branch
}
```

Figure 5.7: Example for a basic block starting at address `0x5a` and translated to C++ by *evm2cpp*. The original EVM basic block is on the left and the translated basic block in C++ annotated with the respective EVM instructions is depicted on the right. Some stack-related opcodes have no direct counterpart in the emitted C++ code. For example, the `DUP1` instruction has been eliminated during translation.

register-based form, where each register is translated to a C++ local variable, and no register is reused. This is similar to the single-static assignment form, albeit without the need for PHI nodes. At the beginning and the end of each translated basic block, we ensure that the stack effects of the register-based form and the original EVM opcodes are the same. Essentially, we use the EVM stack only to pass parameters between translated basic blocks. This enables us to eliminate a number of costly EVM stack operations. Furthermore, we emit C++ code that can be well optimized by recent clang versions (we tested *clang* ≥ 13).

Figure 5.7 shows an example of a translated basic block. Here, the `DUP1` opcode is eliminated, which duplicates a value on the EVM stack. Furthermore, we eliminate the `PUSH2` instruction, which pushes a constant to the stack. Owing to the constant propagation pass we perform, we can infer that this constant is used as a jump target later. Instead of dispatching the jump via the EVM stack, we emit a *goto* statement that directly targets the desired block. Before the jump, we fix up the EVM stack effects of the basic block by pushing the right values to the stack. With respect to the EVM stack, the original bytecode performs three pushes, two pops, and one replacement of the top element. In contrast, the generated C++ code uses only a single stack push.

EVM-level Auto-dictionary To increase fuzzing efficiency, many fuzzers scan the code for constants and create a dictionary of potentially interesting values (e.g., file format header magic values). However, they typically scan for up to 64 bit constants or null-terminated C strings and thus do not properly handle the 256 bit EVM words or 160 bit Ethereum addresses. Hence, we generate a dictionary of values based on the constants discovered during the constant propagation pass. This includes computed quasi-constants that are often found in EVM bytecode.

Dynamic Contract Creation Currently, EF↯CF does not support fuzzing contracts that create other contracts at runtime, as EF↯CF would have to execute previously unknown EVM bytecode, which is not possible in the current ahead-of-time compilation model. When EF↯CF encounters an instruction that creates a new contract, EF↯CF will stop executing the current transaction sequence. However, we only encountered a single contract that created a new contract at runtime during our evaluation. As such, we believe these to be sufficiently rare cases. There are multiple options to handle dynamic contract creation. Currently, the ahead-of-time executed code is separated from the interpreter. However, since both use the same data structure to store EVM state, one could seamlessly switch between interpretation and compiled smart contract code. This would reduce fuzzing throughput but would allow EF↯CF to handle dynamic contract creation. If the constructor code of the contract is known, it can also be translated with *evm2cpp*, allowing the fuzzer to execute constructor code at high throughput. Alternatively, one could switch to a cached just-in-time compilation model of the EVM bytecode instead of using ahead-of-time compilation. However, both with just-in-time and ahead-of-time compilation, one needs to handle the fact that some contracts change constants within the constructor code before executing it, which hampers efficient caching of translated code.

5.3.2 Fuzzing Harness

We opted for a lightweight EVM implementation as the base for our fuzzing-optimized EVM runtime. To this end, we adapted the open-source *eEVM* project [eEVM] such that it fits the code-generation of *evm2cpp* and added an implementation of several newer EVM opcodes, missing features, and various minor fixes. Furthermore, we replaced the usage of C++ exceptions with return values in hot code paths that deal with exceptions thrown by the smart contract code. This results in considerably better performance since fuzzing tends to frequently exercise the error handling paths of the smart contract code. We also switch to a more optimized hashmap implementation [Pop19] and use *mimalloc* [LZM19] as the default allocator.

Input Format Within the *eEVM* project, we created a standard *libfuzzer*-compatible fuzzing harness. The bug oracles are directly integrated into the fuzzing harness and runtime support code of the *eEVM* project. The fuzzing harness features a parser for a custom input format we developed. This format can be quickly parsed without ever failing, i.e., any unneeded data is discarded by the harness; for any missing data fields, default values are assumed. This robust parsing approach allows the use of standard bit-flipping mutations [Fio+20] that are unaware of the input structure. The input format consists of an initial header defining the initial environment, followed by a sequence of transactions. Each transaction is represented as a header and a field that specifies the length of the transaction input. Mutating the header for a transaction allows the fuzzer to select transaction-specific parameters, such as the sender account, the call value, and the number of allowed reentrant transactions. For the input data, the parser simply consumes bytes from the fuzzer-provided data according to the length specified in the header until the end of the fuzzer-provided data. Figure 5.8 shows an example for a test case generated by EF↯CF to exploit the contract depicted

in Figure 5.2. We designed the input format to enable high throughput fuzzing while being expressive enough to model complex smart contract interactions. Furthermore, we use the input format as a template to synthesize Solidity attack smart contracts that exploit the target. For each attacker-controlled account, we synthesize a Solidity contract that implements the behavior as specified by the generated test case. Note that we perform a straightforward translation here, which means that the synthesized contracts do not react to the victim contract, but simply perform the calls as they are specified in the test case. The composition of attack contracts implements one big state machine, with a central attack contract as an entry point that synchronizes the state of all attack contracts.

```yaml
number: 0
difficulty: 0
gas_limit: 0
timestamp: 0
initial_ether: 14000000000000000000
txs:
  - sender_select: 1
    call_value: 9227875636482146304
    input: "0xd0e30db0" # deposit()
    returns: []
  - sender_select: 1
    call_value: 0
    input: "0x5fd8c710" # withdrawBalance()
    returns:
      - value: 1
        reenter: 2
        data_length: 0
        data: "0x"
  - sender_select: 1
    call_value: 0
    # addAllowance(0xc3cf2af7ea37d6d9d0a23bdf84c71e8c099d03c2,
    #              1117873197643827594651545771110674982630890210242)
    input: "0xf3c40c4b0000000...."
    returns: []
  - sender_select: 2
    call_value: 0
    # transferFrom(0xc2018c3f08417e77b94fb541fed2bf1e09093edd,
    #              295147905179352825856)
    input: "0x01c6adc30000000...."
    returns: []
  - sender_select: 2
    call_value: 0
    input: "0x5fd8c710" # withdrawBalance()
    returns:
      - value: 1
        reenter: 0
        data_length: 0
        data: "0x"
```

Figure 5.8: Textual representation of the transaction sequence generated by EF/CF to exploit the contract from Figure 5.2 (some fields omitted and simplified.

Fuzzer and Harness Integration While the fuzzing harness itself is mostly oblivious to the used fuzzer, we opted to rely on AFL++ [Fio+20] as one of the most advanced general-purpose fuzzers. AFL++ supports various modern fuzzing techniques, such as collision-free coverage bitmaps, a Redqueen [Asc+19b] implementation called *cmplog* to bypass fuzzing roadblocks, and support for custom mutators. Due to our transpilation approach, we are able to directly leverage the instrumentation of AFL++ for smart contract code. We built the fuzzing harness with *clang*, with the highest optimization setting and link-time optimization (LTO) enabled, instrumenting the harness with AFL++'s LTO-based collision-free code coverage instrumentation. Since we have translated the EVM bytecode to C++ code, we can utilize AFL++'s coverage instrumentation to instrument the combination of the harness and all the transpiled smart contracts.

However, we noticed several issues when using AFL++, most importantly regarding the implementation of the Redqueen mutations [Asc+19b]. The problem is that by default, the optimized big integer library used by *eEVM* uses branchless code when comparing the four 64 bit words that make up a single 256 bit value. As a result, AFL++'s *cmplog* cannot detect when only one of the four words matches, as no new code coverage can be observed. As a consequence, it fails to incrementally solve comparisons with large constants. However, this issue is only relevant for bypassing comparisons and not during other arithmetic operations. As such, we opted to manually adapt the relevant functions to provide explicit coverage feedback to AFL++. This optimization allows AFL++'s *cmplog* to solve a considerable amount of fuzzing roadblocks due to integer comparisons. To further increase fuzzing efficiency when applying structural mutations in the custom mutator, we added a lightweight tracing mode for certain opcodes (comparisons and returns) to the codebase. This enables us to identify quasi-constants and add them to the dictionary of our mutator at runtime.

When fuzzing for reentrancy attacks, we found it beneficial to notify the fuzzer about the call depth of the current execution. To this end, we introduce call-depth-sensitive coverage reporting in the fuzzing harness. Whenever a new basic block is executed, we record the current call depth in AFL++'s coverage map. This allows AFL++ to distinguish executions of the same contract at different call depths. Since AFL++ receives a new coverage signal when a transaction is executed in a reentrant manner, the test case will be stored in the queue. In turn, this increases the probability of finding reentrancy attacks.

Multi Target Selection We have also implemented an experimental multi-target mode in the fuzzing harness, which allows EF/CF to fuzz multiple contracts at once. This is useful in case there are mutual interdependencies between two contracts. Essentially, for each transaction, we allow the fuzzer to choose the target smart contract out of a list of contracts, producing a transaction list that alternates between calling different target contracts. However, we have not implemented support for multiple ABIs in the custom mutator. As a workaround, we concatenate the ABIs of all involved contracts. Nevertheless, the fuzzing efficiency of this mode is not as good as with full support in the custom mutator. We leave optimizing the custom mutator for multi-target ABI-based fuzzing as future work. While some known attacks require such a mode, the vast majority of known vulnerabilities do not fall into this category.

5.3.3 Custom Mutator

We implemented a mutator library, called *ethmutator*, in roughly 10 kloc of Rust to efficiently generate and mutate: (1) the structured transaction input expected by the smart contract code, and (2) the transaction sequence input format parsed by the fuzzing harness. The mutator library features a parser and emitter for the binary input format accepted by the harness code. Based on this library, we implement several related tools, such as a structured test case minimizer and an AFL++ custom mutator. The mutator is carefully engineered with high performance in mind. We reduce the number of required allocations and copy operations by applying copy-on-write semantics while performing mutations on a transaction sequence. We also use *mimalloc* [LZM19] in the custom mutator, as this increases the performance of the mutator by a factor of four.

In Ethereum, a transaction is associated with an input field, which is simply a variable-length byte string. Smart contracts use a de-facto standardized ABI format, which specifies how function calls with parameters of complex types are encoded. To enhance the efficiency, we use the ABI information in the *ethmutator* to perform mutations according to the ABI. Unlike existing general-purpose fuzzers, our custom mutator can handle the complexity of the ABI format by acting as a grammar fuzzer for the given ABI and generating structurally-valid inputs based on the ABI. When choosing the values for primitive types, we rely on a fuzzing dictionary built into the custom mutator. Recall that this dictionary is seeded with the constants that are discovered during the analysis pass of *evm2cpp*. In addition, we extend the dictionary with "interesting" values that are likely to trigger bugs (e.g., the dictionary contains the maximum value for every integer type supported by Solidity to make it more likely to trigger integer overflows). When no ABI is available, we exploit the fact that ABI-encoded data is always similarly structured for efficient mutations. For example, when appending a new transaction, we first select a 4 byte constant from the dictionary as a prefix for the input. Since the basic unit of the ABI is a 256 bit EVM word, most of the input mutations operate on this word size if no ABI is available.

The second main task of the custom mutator is to apply structured mutations to the transaction sequence. We implemented several basic structured mutations, such as adding, duplicating, or dropping transactions. Furthermore, we implemented more involved mutators, such as structural test case splicing or value propagation between transactions in a sequence. Whenever the base fuzzer adds a test case to its queue, the custom mutator parses this test case and keeps the transaction sequence in memory. The structured splicing mutation then replaces a randomly selected transaction sub-sequence with a sub-sequence obtained from a previous test case. The intuition here is that transaction sequences from prior test cases contain valid transaction combinations. Combining two valid transaction sequences is more likely to result in a new valid transaction sequence. We also propagate values from earlier transactions to later transactions. Hence, with some probability, values in the transaction input will be replaced with values that occurred as parameters in the input of earlier transactions. Similarly, we set the value of *address* types in the ABI to the address of attacker-controlled accounts that previously already sent a transaction. Similar to AFL [Zal], the custom mutator has multiple stages and a fixed set of mutations applied to every

test case. Subsequently, random mutations are applied (i.e., similar to the *havoc* phase in AFL). Finally, the custom mutator also uses stacked mutations, where different random mutation operations are combined.

En/Decoding the ABI We use the *ethabi* Rust library to parse the JSON-based ABI definition, which allows us to en- and decode the ABI format expected by the smart contract. Sometimes the base fuzzer will break the ABI encoding, which results in the custom mutator attempting to decode extremely large input data. In fact, during the development of EF↯CF, we uncovered and fixed two bugs in the *ethabi* library, s.t., it would not panic when attempting to decode broken ABI-encoded data. However, we still set an 8 kbyte limit to the number of bytes we attempt to decode. This prevents EF↯CF from spending too much time attempting to decode an unusually large input byte-string of a transaction. Since it is unlikely a valid or useful input for the smart contract under test, it is preferable to avoid the lengthy decoding process altogether. In these pathological cases, we fall back to the random mutations provided by the base fuzzer.

AFL++ Integration We patched AFL++ for optimal integration with our custom mutator. Our patches make AFL++ report an internal performance score to our custom mutator. The custom mutator can then adapt this score to select the number of fuzzing rounds for a given test case. Depending on the size and complexity of the test case, we apply different types of mutations and a different number of fuzzing rounds in the custom mutator.

We ensure that AFL++ extensively uses the structured trimming provided by our custom mutator. We found that structured trimming is beneficial to the fuzzing process in EF↯CF. Additionally, we also utilize the trimming step of AFL++ to update the internal test case queue of our custom mutator. This allows us to only add test cases to the internal queue, which are trimmed. In contrast to AFL++'s queue, we keep the internal queue completely in memory and use it for efficient structural splicing operations.

We also extend AFL++ with an additional manual feedback API with a function that allows reserving a larger part of the AFL++'s coverage map for direct feedback. This is used by our modifications to the *eEVM* runtime to provide explicit feedback on two properties of the execution. First, we provide explicit feedback on the progress of solving comparison operators. For example, for the `EQ` opcode, we provide explicit coverage feedback to AFL++ for every 64 bit part of the 256 bit EVM-native integer that is equal. This allows AFL++'s *cmplog* mode, which implements input-to-state correspondence on the 64 bit comparison level, to effectively solve fuzzing roadblocks on EVM-native 256 bit level. Similarly, we provide explicit coverage feedback to AFL++ whenever a contract executes in a reentrant call. This allows the fuzzer to distinguish a reentrant execution from a normal execution, i.e., a simple form of context-sensitive coverage. We found this beneficial for AFL++ to keep both reentrant and non-reentrant variants of the same transaction sequence in the queue.

When launching AFL++, we disable the byte-level *auto-dict* feature since it is superseded by our replacement acting on the EVM bytecode level. Re-implementing

the *auto-dict* feature in *evm2cpp* results in significantly smaller and more useful dictionaries than relying on AFL++'s auto-dict mode, which scans the final binary for constants and also picks up irrelevant data, such as strings internal to the *eEVM* runtime.

Multi-Core Fuzzing While AFL++ has a single-threaded design, it is capable of synchronizing with other instances of AFL++ (and even other fuzzers) via the filesystem. EF/CF inherits the same technique, and the wrapper scripts we provide as part of EF/CF can automatically launch multiple AFL++ instances. Generally, it is recommended to launch AFL++ using multiple different configurations when using multiple cores [Fio+20]. We adapt these recommendations to EF/CF. When running on four or more cores, we launch the following configurations:

1. A main instance with AFL++'s deterministic mutation stages enabled.
2. A compare solver instance, with input-to-state [Asc+19b] and EVM-level compare tracing enabled.
3. One instance fuzzes only with the custom mutator.
4. The remaining cores utilize AFL++'s lightweight havoc mutations and our custom mutator.

Other Bug Oracles The standard bug oracles are implemented inside of the *eEVM* runtime code. To implement a new bug oracle, one has to modify the C++ implementation of the runtime code. For performance reasons, EF/CF does not rely on heavyweight program analysis techniques such as taint tracking to implement bug oracles. As such, the bug oracles in EF/CF are limited to detecting bugs based on the state of the simulated Ethereum blockchain. However, we believe that the existing bug oracles supported by EF/CF already cover a large set of use cases. Furthermore, developers can use custom properties or the event mechanism to implement custom bug oracles directly in Solidity code.

Currently, EF/CF supports optional fuzzing modes, which are also used by industry fuzzers such as Echidna [Gri+20] or Mythril [Conc; WC20]. For example, EF/CF supports property-based fuzzing with an interface that is fully compatible with the Echidna fuzzer. The developer specifies a property of the contract that must always hold, i.e., an invariant of the contract code. Such properties are specified as a Solidity function that returns whether the property is currently true or false. After every executed transaction, EF/CF calls the configured property functions and checks whether the return value signals a violated property. Similarly, EF/CF can utilize the EVM event logging and error propagation mechanisms to detect bugs. The smart contract developer emits a certain event whenever a bug is triggered. Whenever this event is logged during fuzzing, EF/CF will consider the execution to trigger a bug and report it. Similarly, Solidity versions 0.8 or above report special error messages to the caller whenever an assertion is violated or an integer overflow happens. If configured, EF/CF picks up these special error message return codes as a bug and reports it. This way, EF/CF can be utilized to fuzz for more than Ether-based bugs and also uncover contract-specific logic bugs.

Comparison of Bug Oracles Previous analysis tools often implement a wide variety of bug oracles [Cho+21; Luu+16; Nik+18; Tor+21a] to detect security vulnerabilities, code smells, and other potentially interesting properties of the code. However, the definition of bug oracles and what oracles should be considered as a security vulnerability differ across the literature. We identify the *unprotected selfdestruct* bug oracles as one of the few oracles that are recognized in almost all analysis tools. As such, we utilize this bug oracle in our benchmarks (see Section 5.4). In EF↯CF, we focus on Ether gains as our primary bug oracle, as it features the least number of false alarms in practice. However, this single bug oracle in EF↯CF actually maps to multiple bug oracles in other tools. Furthermore, we implement several additional optional bug oracles that can be used in EF↯CF. We show a comparison of supported bug oracles in Table 5.1. In the following, we discuss some of the bug oracles in more detail.

Locking Ether is fundamentally a liveness property. In general, liveness properties are hard to show with a fuzzer. A standard fuzzing approach can only show that a certain code path *can* be reached. However, to accurately report locked Ether, the fuzzer would have to show that a certain code path cannot be reached. For this reason, many fuzzers actually implement a static analysis approach to detecting locked Ether. For example, ILF [He+19] and Confuzzius [Tor+21a] simply scan the contract for any instruction that can—in theory—send Ether. However, they do not verify that this instruction can be actually executed, i.e., the instruction could be inside of dead code. As such, this approach will only detect simple cases of locked Ether.

Leaking Ether and *Ether Gains* are two very related bug oracles. Both attempt to identify bugs where the contract can be used to send Ether to some unrelated address. EF↯CF supports detecting leaking Ether but disables the bug oracle by default. The idea is that the attacker can trick the contract into sending Ether to some contract that has no previous relationship with the contract. However, many contracts support transferring Ether indirectly to another address as a feature. For example, all token contracts must support transferring tokens to arbitrary addresses as a feature. In contrast, EF↯CF uses Ether gains as a bug oracle that covers more realistic cases. EF↯CF will report an Ether gain bug whenever the sum of the Ether balances of all attacker-controlled accounts exceeds the initial sum of balances. This allows EF↯CF a wider and more realistic set of issues. For example, the vulnerability depicted in Figure 5.6 is neither detected by Confuzzius [Tor+21a] nor Smartian [Cho+21]. However, since EF↯CF simulates multiple attacker-controlled accounts, it will quickly generate a transaction originating from the first account and leaking the Ether to a second attacker-controlled address.

Most analysis tools feature explicit *Reentrancy* bug oracles. In contrast, EF↯CF does not feature an explicit detector for reentrancy but simply generates reentrant transaction sequences that trigger other bug oracles, such as Ether gains. As such, EF↯CF detects reentrancy bugs, but only if they are actually exploitable. In contrast to other analysis tools, this leads to fewer false alarms, e.g., when encountering manual reentrancy locking [Rod+19].

By default, EF↯CF reports *unprotected selfdestruct* only if the self-destruct will transfer the remaining Ether of the target contract to the attacker, i.e., the address parameter of the self-destruct is controlled by the attacker. Optionally, EF↯CF can also report *DoS*-style unprotected self-destructs, i.e., if the self-destruct can be triggered by

anyone but always targets a trusted address such as the owner. This style of detection is featured in most other analysis tools.

With Solidity version 0.8, contracts feature automatic integer overflow checking and proper assertion violation reporting. EF/CF supports this new Solidity exception mechanism to signal errors to the fuzzer. Previously, *assert* statements were implemented with the *INVALID* opcode that also triggers a transaction revert. However, earlier contracts (pre 0.4) also used this to implement the failure of input sanitization. As such, it is hard to reliably distinguish between a regularly failing transaction and an assertion violation across solidity versions. We expect developers to use the newer Solidity versions for newly deployed contracts. As such, we opted to support only the new Solidity exception mechanism, which allows us to reliably detect internal errors in a smart contract. This includes memory allocation failure, integer overflows, and internal assertions. Developers can utilize EF/CF for general robustness testing of their newly deployed smart contracts.

Table 5.1: Comparison of bug oracles in various analysis tools with the bug oracles available in EF/CF. ✓ fully supported. × not supported. ✓* supported but not enabled by default. ✓† only supported for contract compiled with Solidity version > 0.8. ×‡ only if it leads to triggering another bug oracle.

Bug Name	EF/CF	Confuzzius [Tor+21a]	Smartian [Cho+21]	Echidna [Gri+20]
Assertion Failure	✓†	✓	✓	✓*
Arbitrary Write	×‡	×	✓	×‡
Block State Dependency	×‡	✓	✓	×‡
Control-flow Hijack (JUMP)	×‡	×	✓	×‡
Custom Event Oracle	✓*	×	×	✓*
Custom Property Checking	✓*	×	×	✓*
Ether Gains	✓	×	×	×
Integer Overflow	×‡ / ✓†	✓	✓	×‡
Leaking Ether	✓*	✓	✓	×
Locking Ether	×	✓	✓	×
Multiple Send	×	×	✓	×
Reentrancy	×‡	✓	✓	×
Require Violation	×	×	✓*	×
Transaction Origin Use	×‡	×	✓	×‡
Transaction Order Dependency	×	✓	×	×
Unsafe Delegatecall	✓	✓	✓	×
Unprotected Selfdestruct	✓*	✓	✓	✓*
Un/Mishandled Exception	×‡	✓	✓	×

Additional Tooling Based on our custom mutator code, we additionally implemented several tools that proved to be useful for smart contract fuzzing. For instance, EF/CF also features a test case minimizer that performs structural minimization on a test case, allowing an analyst to reduce the size of a test case. Furthermore, EF/CF integrates a translator between our binary test case format to a human-readable equivalent *yaml*-based format, allowing for easy manual modification of test cases. EF/CF can also convert a test case into a Solidity attack contract that can be deployed within a blockchain environment to study the generated attack. These tools help an analyst in performing root cause analysis given a test case that triggers a bug.

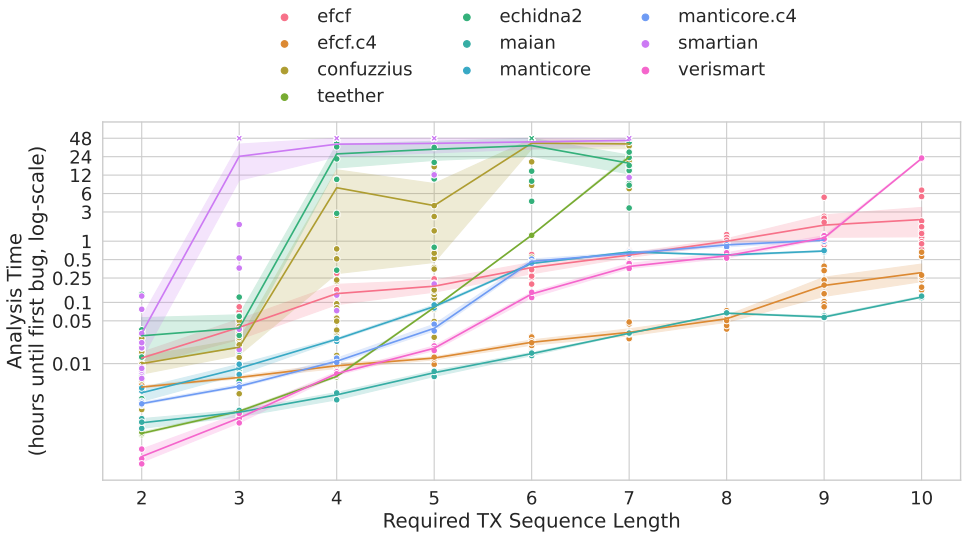


Figure 5.9: Scalability experiment using the synthesized *multi* contract variant that are scaled according to the number of required transactions.

5.4 Performance Evaluation

In this section, we evaluate various performance aspects of EF↯CF and compare EF↯CF to the current state-of-the-art in fuzzing and symbolic execution. We evaluate EF↯CF concerning scalability to longer transaction sequences, the test case throughput, scalability to multiple cores, and achieved code coverage. We also perform an ablation study that shows how the various components of EF↯CF improve the performance of the overall fuzzer.

5.4.1 Scalability Benchmarks

We start with an evaluation of the effectiveness of analysis tools in dealing with an increasing length of transaction sequences. To this end, we created a benchmark consisting of three types of contracts (*multi*, *complex*, and *justlen*), which model different code structures and roadblocks (that hinder analysis) typically found in smart contracts. For each type of contract, we devise several variants (9 for *multi*, 3 for *complex*, and 4 for *justlen*) that require an increasing number of transactions to reach an exploitable state plus another transaction to trigger a vulnerability (see Table 5.2 for a summary). Each variant of *multi* and *complex* contracts is parameterized given the number of transactions that are needed to trigger the bug. For instance, contract $multi_{10}$ requires 10 transactions (or sequential function calls) to reach an exploitable state. Here, we chose to insert a vulnerability in a function that simply triggers *selfdestruct* of the contract when the exploitable state is reached. This type of bug is widely supported by analysis tools and allows us to compare various tools according to the analysis time required to identify the bug.

The *multi* contracts are synthesized as follows: for each function, several equality or in-equality constraints are enforced on up to six integer arguments before setting a

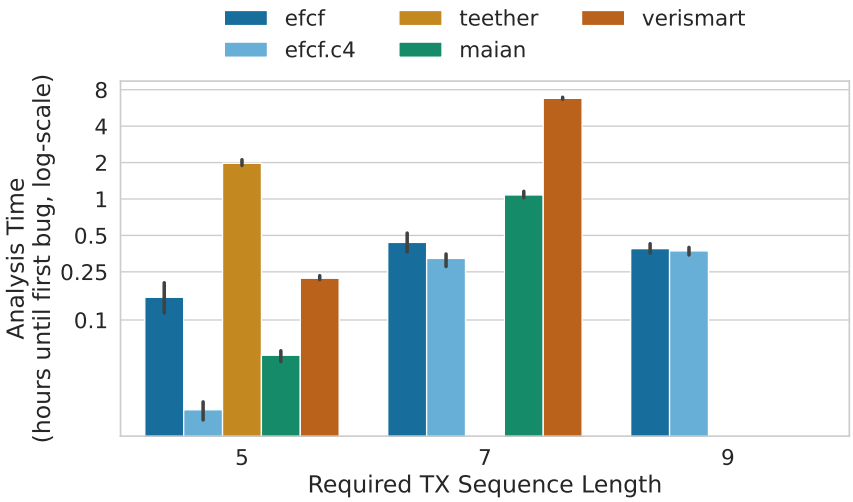


Figure 5.10: Scalability experiment using the manually created *complex* contract variant, scaled according to the number of required transactions.

boolean internal state variable. As such, these benchmark contracts test the capability of solving input constraints across multiple transactions. The functions of the contract must be called in the right order, with the right inputs, to trigger a *selfdestruct* of the contract. Note that these contracts favor symbolic execution tools, given that they do not contain any control-flow statements except for error handling. As such, there is no potential for path explosion within the functions. The three *complex* contracts consist of a manual adaptation of the *multi* contracts with more varying constraints on the input and the internal state of the smart contract (e.g., requiring an array input of a certain length or requiring the *sha3* hash of a fixed value as input). Finally, the *justlen* example is adapted from Groce et al. [GG21] and is parameterized over the length of an array that must be reached using *push*, *pop*, *double*, and *halve* operations.

We run the tools tEther [KR18a], MAIAN [Nik+18], EthBMC [FAH20], Manticore [Mos+19], Confuzzius [Tor+21a], and Echidna [Gri+20; GG21] using this setup and measure the time until the bug is discovered with a global timeout of 48 h. We run all analysis tools within Docker containers on an Intel Xeon Gold 6230 CPU clocked at 2.10 GHz with 188 Gbyte RAM. Due to time constraints, we run the tools in parallel, keeping all physical CPU cores fully occupied. We do not utilize hyperthreaded cores, as they are detrimental to CPU-bound tasks such as fuzzing. By default, all tools were executed on a single core. We additionally ran those tools that support multi-core analysis (EF↯CF and Manticore) on 4 cores concurrently. To run this experiment, we had to patch the tools tEther [KR18a], MAIAN [Nik+18], and ConFuzzius [Tor+21a] such that they support longer transaction sequences. We excluded EthBMC [FAH20] from most of our experiments since we were not able to identify the bug in EthBMC, which causes it to not report any vulnerabilities with transaction sequences longer than 3. Moreover, we excluded ILF [He+19] because a machine learning-based fuzzer is very unlikely to produce the 256 bit magic value constants required for the synthesized contracts. When possible, we bound the number of transactions to consider by 32 for

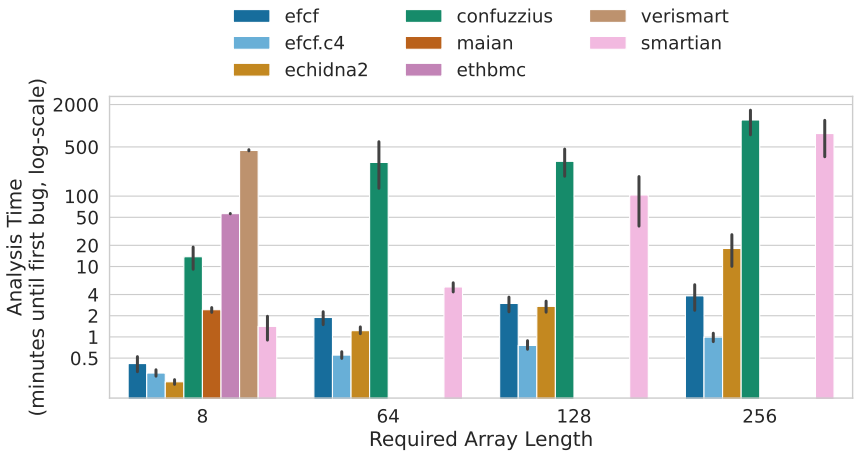


Figure 5.11: Scalability experiment using the *justlen* contract by Groce et al. [GG21], scaled according to the length of the array that must be created using push, pop, halve and double operations.

all tools in order to ensure that they execute at a reasonable pace while leaving enough room for failing/duplicated transactions within sequences. We run all Python-based tools with CPython and the PyPy JIT-compiler. We report the numbers with PyPy for MAIAN and teEther, where we observed a speed-up. We perform 3 trials for all symbolic execution tools (where randomness does not play a big role) and at least 10 trials for all fuzzers. In total, we spent approximately 970 days of CPU time for this experiment.

Our results are summarized in Table 5.2. EF$\lightning$CF is the only analysis tool capable of solving all of the contracts in this benchmark dataset. Figure 5.9 shows the log-scaled time required to solve the *multi* contracts with an increasing number of required transactions. We omit those tools/results from the plots where all runs fail to identify a bug. Only three of the tools scale reasonably to a larger number of transactions: Manticore, EF$\lightning$CF, and MAIAN. Surprisingly, MAIAN performs best in this benchmark and seems to scale very well. Manticore scales in a similar manner as EF$\lightning$CF, except for the longest transaction sequence, where Manticore fails to identify the vulnerability. Surprisingly, Confuzzius, using hybrid fuzzing and concolic execution, performs worse than most pure symbolic execution-based tools on this benchmark. Similarly, the fuzzer Echidna does not perform well on the *multi* and *complex* contracts—highlighting the limitations of Echidna in solving complex input constraints on multiple transactions.

Notice that most analysis tools that we considered in this experiment are inherently single-threaded. In contrast, it is straightforward to parallelize EF$\lightning$CF. Running EF$\lightning$CF on 4 cores in parallel results in a significant speed improvement as can be seen in Figure 5.9 and in more detail in Section 5.4.4.

With the increased complexity of the input constraints, symbolic execution tools lose their edge, as can be seen in Figure 5.10. Here, EF$\lightning$CF is the only tool that solves all *complex* contracts within a reasonable time. The *justlen* benchmark contracts use loops to push and pop values of an array and, as such, favor fuzzing-based tools when

compared to symbolic execution tools that provide limited effectiveness due to path explosion (see Figure 5.11). Note that EF↯CF's analysis time is almost constant for this benchmark. This is due to the fact that EF↯CF features a mutation operation that duplicates transactions to become reentrant transactions. We designed this mutation to quickly identify same-function reentrancy bugs, but incidentally, this is also the reason why EF↯CF can very quickly solve *justlen* contracts. At some point, EF↯CF will simply duplicate the transaction, which doubles the array size. As such, there is nearly no difference in reaching an array size of 64 or 128.

Table 5.2: Capability of analysis tools to identify bugs with increasing transaction sequence length. ✓ bug can be found, × bug never found within 48 h.

Tool	multi					complex			justlen			
	2	3-7	8	9	10	5	7	9	8	64	128	256
teEther [KR18a]	✓	✓	×	×	×	✓	×	×	×	×	×	×
MAIAN [Nik+18]	✓	✓	✓	✓	✓	✓	✓	×	✓	×	×	×
EthBMC [FAH20]	✓	×	×	×	×	×	×	×	✓	×	×	×
Manticore [Mos+19]	✓	✓	×	×	×	×	×	×	×	×	×	×
ConFuzzius [Tor+21a]	✓	✓	×	×	×	×	×	×	✓	✓	✓	✓
Echidna [Gri+20]	✓	✓	✓	×	×	×	×	×	✓	✓	✓	✓
VeriSmart [SHO21; So+20]	✓	✓	✓	✓	✓	✓	✓	×	✓	×	×	×
Smartian [Cho+21]	✓	✓	×	×	×	×	×	×	✓	✓	✓	✓
EF↯CF	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

5.4.2 Scalability Ablation Study

As part of our ablation study, we also compare the various configurations in terms of scalability to longer and more complex transaction sequences. We utilize the same scalability experiment as in Section 5.4.1. However, we bound the execution time to a maximum of 8 h. We run EF↯CF in four configurations: Full EF↯CF with ABI, full EF↯CF without knowledge of the ABI, fuzzing with the custom mutator only (*EM*), and fuzzing with AFL++'s mutations only (*AFL*). The results are shown for the *multi*, *complex*, and *justlen* benchmarks in Figure 5.12, Figure 5.13, and Figure 5.14, respectively. While the *AFL* configuration generally provides the highest throughput (see Table 5.3), it lacks structured mutation operations, such as splicing at the transaction level. As such, the fuzzing process becomes ineffective and fails to reliably generate even short meaningful transaction sequences. Furthermore, the *AFL* configuration fails to identify a bug in any of the *complex* variants. This shows that the custom mutator in EF↯CF is essential to good fuzzing performance. Furthermore, we can see that in this benchmark that the *EM* configuration performs best. The magic value comparisons are best solved using the dictionary-based sampling employed by the custom mutator for integer types. As such, spending more fuzzing time on dictionary sampling increases the probability that the right value is sampled from the dictionary. However, since the custom mutator utilizes the dictionary probabilistically, we also observe some extreme outliers in these experiments. As the custom mutator

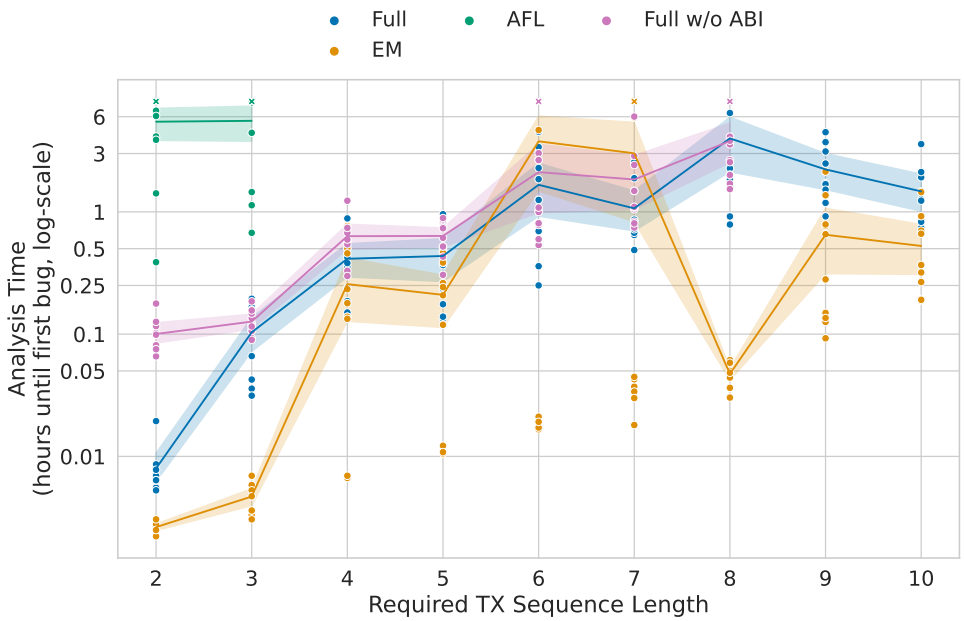


Figure 5.12: Scalability experiments on various configurations of EF/CF with the *multi* contact.

performs mostly structural mutations on the transaction sequences, we can see that it also performs best on the *justlen* experiment.

While the *EM* configuration performs best on the benchmarks presented here, we found that without AFL's mutations, and especially the input-to-state correspondence [Asc+19b] mutation, there are several fuzzing roadblocks in practice, which cannot be solved by the *EM* configuration. Furthermore, the benchmarks focusing on real-world smart contracts in Section 5.4.3 show that the standard EF/CF configuration performs best.

Fuzzing without ABI When fuzzing without ABI information, EF/CF fully relies on the coverage feedback to discover useful transaction inputs. Generally, EF/CF without ABI information expectedly performs worse than fuzzing with ABI information. On the *multi* and *complex* contracts fuzzing without ABI information identifies the bug in a mean of 223 min ($\sigma = 205$). In comparison, fuzzing with ABI information requires a mean of 87 min ($\sigma = 127$) to identify the bug. On the *justlen* benchmark, the difference is much smaller: 3.9 min with ABI and 6.6 min without ABI. The fuzzer does not need to identify correct input parameters to identify a bug in the *justlen* benchmark, as most of the exposed functions simply do not require parameters. Performing structural mutations on the transaction sequence does not require ABI information, and as such, the performance difference is much smaller.

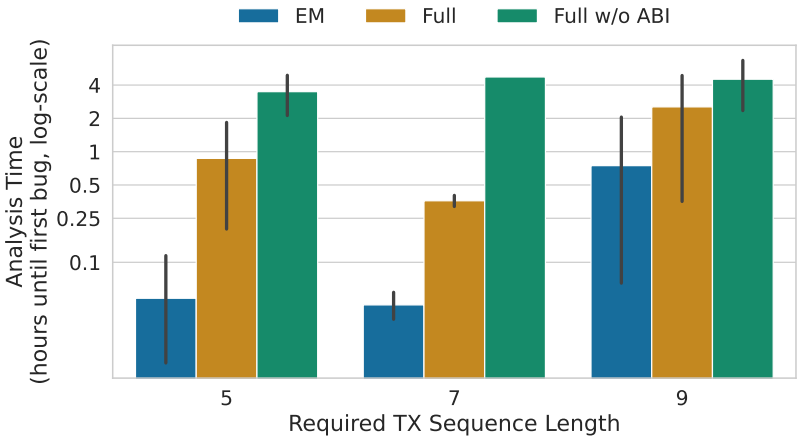


Figure 5.13: Scalability experiments on various configurations of EF/CF with the *complex* contact.

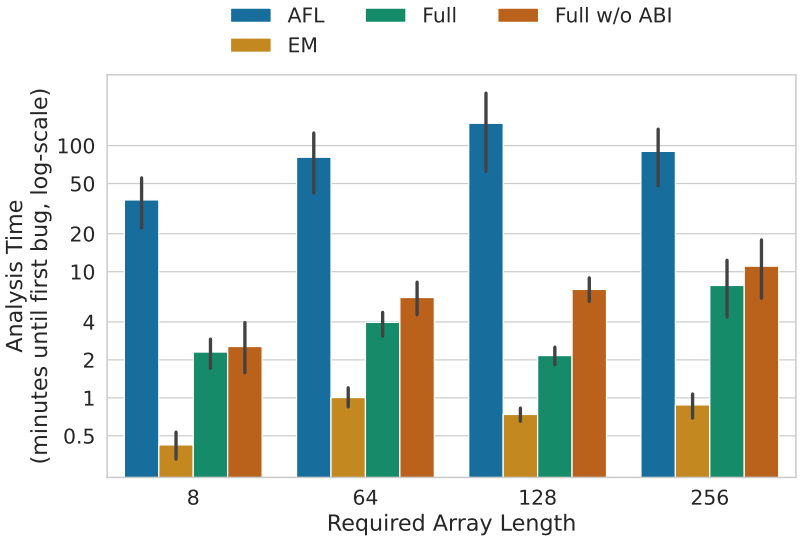


Figure 5.14: Scalability experiments on various configurations of EF/CF with the *justlen* contact.

5.4.3 Throughput Ablation Study

We evaluate the various components of EF↯CF and how they affect test case throughput and achieved code coverage. We leverage a set of contracts consisting of a mix of real-world and one of our benchmark contracts: the contracts Crowdsale [He+19] and the synthetic multi_{10} are representative of simpler contracts, while the IMBTC [imBTC], SpankChain [SpCLC], CryptoBets and PackSale contracts are more complex real-world contracts. We run each fuzzing configuration 20 times for 10 minutes and record the average test case executions per second along with the EVM basic block coverage. Our evaluation results are summarized in Table 5.3. We provide further experiments with the *multi*, *complex*, and *justlen* benchmark contracts in Section 5.4.2.

We observe a significant drop in test case throughput when we disable *evm2cpp* and run the smart contract in the EVM interpreter (designated as *Interp* in Table 5.3). This directly translates to achieving less code coverage with the fuzzer, highlighting the importance of *evm2cpp* in our design.

For the *AFL* configuration, we disable the custom mutator. For the *EM* configuration, we disable AFL's mutations. While the lightweight mutations performed by AFL++ result in the highest throughput, they are not aware of the input structure and often achieve worse code coverage given a certain time budget. Furthermore, we noticed that AFL++ alone fails to create combinations of transactions (see Section 5.4.2). In this respect, basic block coverage is not a good metric since it does not account for different combinations of transactions. Similarly, the custom mutator alone sometimes does not discover all code paths due to worse throughput and the lack of AFL++'s input-to-state correspondence. Typically, a combination of both, i.e., running full EF↯CF, results in the best code coverage. Since fuzzing campaigns utilize multiple cores in practice, we take inspiration from *ensemble fuzzing* [Che+19b] and automatically launch different fuzzer configurations in parallel if multiple cores are available (see Section 5.3).

We compare the throughput of other fuzzers to that of EF↯CF. Our measurements show that the test case throughput of EF↯CF is larger by one order of magnitude when compared to other fuzzers. Within our benchmark set, EF↯CF has a mean throughput of 24 301 exec/s ($\sigma = 7154$). Echidna [Gri+20] achieves a throughput of 189 ($\sigma = 183$) and a maximum throughput of 497 test cases per second. Confuzzius [Tor+21a] has a mean throughput of 78 transactions per second ($\sigma = 25$). The transaction throughput is always higher or equal to test-case throughput since test cases typically consist of multiple transactions.

Table 5.3: Throughput measurements in average test case executions per second and mean code coverage. Best coverage is underlined.

		Interp		AFL		EM		Full	
Contract	LOC	exec/sec	cov %	exec/sec	cov %	exec/sec	cov %	exec/sec	cov %
Crowdsale	41	13042	79.5	29688	76.1	19868	80.6	25858	<u>87.3</u>
multi_9	150	12814	43.7	43053	40.8	20641	<u>75.7</u>	25817	52.5
IMBTC	664	6880	36.6	28372	52.6	18444	36.4	34510	<u>52.9</u>
PackSale	730	7146	65.0	32215	67.5	11952	60.2	24672	<u>75.2</u>
Spankchain	1048	6574	26.9	19837	47.0	17245	<u>50.5</u>	22698	41.7
CryptoBets	1142	3174	30.8	25122	35.0	10256	<u>45.0</u>	12246	40.5

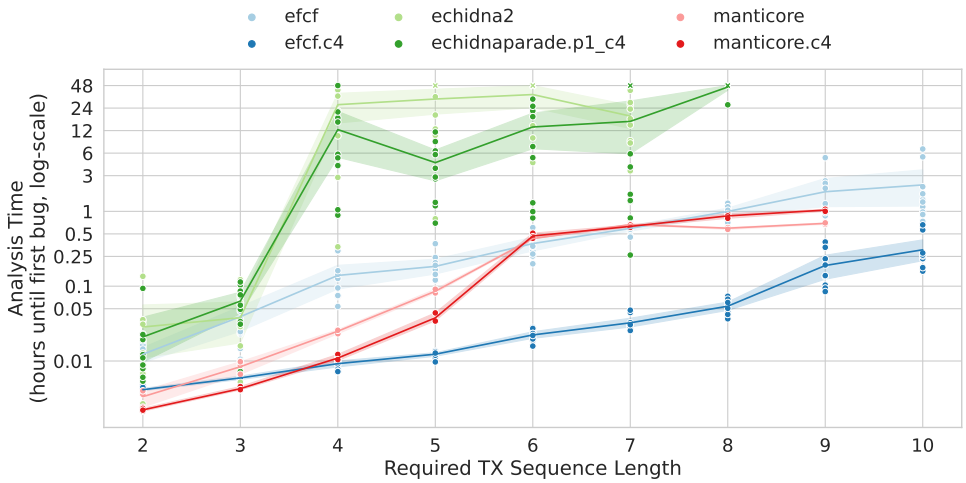


Figure 5.15: Results of running multiple analysis tools on a single core vs. running on 4 cores (marked with *c4*) in parallel on the *multi* dataset.

5.4.4 Multi-Core Performance

We also evaluate the multi-core performance of those analysis tools that support it: EF/CF, Manticore, and Echidna. To parallelize Echidna, we utilize the *echidna-parade* [GG21] tool to run multiple instances of Echidna in parallel. In contrast to the normal mode of operation in *echidna-parade*, we always fuzz the full set of functions by not excluding any function from the fuzzing runs. In our benchmarks, the default mode of operation is detrimental to performance in terms of time-to-bug. Manticore natively supports multi-threaded analysis to leverage multiple cores. For EF/CF, we leverage the multi-core fuzzing approach of AFL++.

Figure 5.15 shows the multi-core performance of several analysis tools on the *multi* contracts. We can see that with EF/CF, the performance significantly increases between the single and multi-core versions. This is primarily because EF/CF utilizes an approach similar to ensemble fuzzing that spawns EF/CF's core fuzzer in multiple different configurations. Similarly, parallelizing Echidna with the *echidna-parade* tool shows significant improvements over the single-core Echidna. In a multi-core setting, Echidna can find the transaction sequences up to 8 transactions and features a significant speed-up for the transaction sequences with lengths 4 to 7. For both fuzzers, we observe that some single-core runs are as fast or faster than other multi-core runs. This is because of the probabilistic nature of fuzzing. However, the multi-core runs reduce the variance between the runs, allowing the fuzzer to identify the bug in a given time span more consistently. Interestingly, Manticore, the only other tool with built-in multi-core support, does not gain a significant speedup in this experimental setup. We suspect that the symbolic execution approach taken by manticore cannot fully leverage multiple cores.

5.4.5 Code Coverage Comparison

We perform scalability experiments using our own set of benchmark contracts. In this section, we analyze the performance of EF↯CF with respect to code coverage. Note that code coverage is a necessary, but not sufficient, condition to also trigger bugs. We compare EF↯CF to two of the current state-of-the-art fuzzers for smart contracts, namely ILF [He+19] and Confuzzius [Tor+21a]. Instead of using synthetic contracts, we assess the capabilities of these fuzzers on a wide variety of real-world smart contracts. The goal of this experiment is to compare the quality of the fuzzer-generated inputs.

We do not compare the fuzzers according to their time-to-bug, since (1) there are no good datasets available that feature both ground truth and realistically complex contracts, and (2) these fuzzers utilize bug oracles that differ too much to be directly comparable. As such, we restrict ourselves to comparing the quality of the fuzzer-generated inputs according to the achieved code coverage.

To create a realistic and useful dataset for our evaluation, we analyze the *smartbugs-wild* dataset [Dur+20] that consists only of real-world smart contracts. We rank the contracts according to their peak Ether balance as reported by the `etherscan.io` service. We then selected the top 1000 contracts according to this ranking such that we get a realistic set of important smart contracts. We do not sample arbitrarily from the Ethereum blockchain, as this would increase the risk of obtaining toy contracts. However, not all of the contracts are suitable for our comparison. As such, we select a subset of the acquired contracts that are supported by all fuzzers in our evaluation. For example, the contract must not require constructor parameters, as this allows us to easily deploy the contract in all fuzzers we evaluate. In the end, the resulting dataset consists of a set of 253 contracts, which are selected from the top 1000 contracts of the *smartbugs-wild* dataset.

We configure all the fuzzers in this experiment such that they behave reasonably similarly and allow us to compare their results. For example, we noticed that Confuzzius achieves substantially broader coverage whenever a smart contract contains hard-coded addresses. The reason is that Confuzzius starts generating transactions originating from those hard-coded addresses. This is something the other fuzzers do not support. While this behavior does lead to good code coverage, the generated transaction sequences cannot actually be performed on the blockchain, as an arbitrary address cannot be impersonated. As such, we disabled marking the *caller*, i.e., the origin of a transaction, as an unconstrained symbolic value in Confuzzius. Additionally, we had to patch Confuzzius's coverage reporting to take bytecode as input instead of source code. For ILF, we patched ILF to improve the reporting of code coverage and transaction inputs and replaced the existing threshold on the number of generated inputs with a time-based limit. For EF↯CF, we enabled an over-approximating mode, where all external addresses are considered to be contracts under attacker control. Similar to the other fuzzers, we allow EF↯CF to emulate calls originating from the creator of the contract (e.g., the *owner* of the contract with special permissions).

To compare our implementation to existing fuzzers, we follow state-of-the-art recommendations for comparing different fuzzers using statistically sound methods [AB14; Kle+18]. We repeat every experiment 30 times to account for the inherent randomness in the fuzzing process. We limit the runtime to five minutes for each target. Due

to the limited complexity of smart contracts compared to traditional software, this allows most fuzzers to reach a coverage plateau. To calculate the required statistical evaluation metrics, we use SENF [Paa+21b], which allows us to easily assess how each tested fuzzer performs and compare the fuzzers with each other.

The results of our experiments are summarized in Table 5.4a. Our results show that EF↯CF outperforms Confuzzius with statistical significance on 141 targets and on 120 targets when compared to ILF. Confuzzius and ILF perform better than EF↯CF on 83 and 112 of the target contracts, respectively.

We also rank the contracts within our test set according to various complexity metrics. We conduct an additional evaluation on the top 100 most complex targets with respect to the number of logical lines of code, the number of contracts and libraries, the number of comparisons, and the number of branches (see Section 5.1). As shown in Table 5.4b, EF↯CF performs better than Confuzzius as well as ILF on the majority of contracts across all complexity properties. Thus, we conclude that EF↯CF can handle increasingly complex Ethereum smart contracts better than existing fuzzers.

Table 5.4: Comparison of smart contract fuzzers based on code coverage and the SENF [Paa+21a] statistical evaluation framework.

(a) Comparison of all fuzzers on the full test set: the number of times fuzzer A outperformed fuzzer B.

Fuzzer A \ Fuzzer B	EF↯CF	Confuzzius	ILF
EF↯CF	-	141	120
ConFuzzius	83	-	105
ILF	112	136	-

(b) Number of targets, where EF↯CF statistically significantly outperforms ConFuzzius/ILF and vice versa on the top 100 targets in various complexity properties

	EF↯CF : ConFuzzius	EF↯CF : ILF
#LLOC	73 : 17	55 : 37
#funcs	66 : 22	46 : 42
#comp	60 : 28	66 : 26
#branch	77 : 15	54 : 37

5.5 Bug Detection Capabilities

To assess the bug detection capability of EF/CF, we compare EF/CF with real-world contracts that we obtained from prior studies [Bos+22; Cec+21; FAH20; Rod+19; Zho+20]. Note that after initial analysis, we concluded that existing benchmark datasets with ground truth for Solidity/Ethereum analysis tools [Dur+20; GP20] are not suitable for testing/comparing dynamic analysis tools such as fuzzers. Both benchmark datasets consist mostly of rather simple and very similar contracts. For example, the curated reentrancy dataset of Durieux et al. [Dur+20] features mostly honeypot contracts that are designed to be easily analyzed [TSS19]. Similarly, the reentrancy bugs injected by Ghaleb et al. [GP20] are too simplistic: many of the injected bugs cannot be triggered by dynamic analysis tools (dead code) or are trivially exploitable. We discuss these issues in more detail in Section 5.5.3. As such, we rely on prior studies on real-world contracts for evaluating the bug detection capabilities of EF/CF. In Section 5.5.1 we show that EF/CF is capable of identifying the vast majority of access control bugs that have been identified in prior studies. Furthermore, EF/CF is able to overcome the limitations of symbolic analyses and identify even more vulnerabilities in those contracts, where the symbolic analyzer *EthBMC* timed out. Furthermore, we assess the capability of EF/CF in discovering several reentrancy attacks known from prior studies in Section 5.5.2. We compare with the symbolic analyzer *Sailfish* and show that EF/CF identifies only those reentrancy issues that pose real security threats. The study by Bose et al. [Bos+22] reports 26 contracts with true reentrancy bugs in the dataset. However, we were able to confirm only 5 of these contracts to be vulnerable to Ether stealing with EF/CF. Our (manual) analysis of the remaining 21 contracts reveals that, while the contracts can be reentered in theory, all but one cannot be exploited. Finally, we show that EF/CF is capable of identifying compositional security issues by evaluating on the contracts used by Cecchetti et al. [Cec+21] in the evaluation of their static analyzer. Furthermore, we show that EF/CF can automatically generate an exploit for the *Uniswap/IMBTC* compositional reentrancy attack given only a list of involved smart contracts.

5.5.1 Access Control Vulnerabilities

Access control bugs such as an unprotected *selfdestruct* have been widely investigated [FAH20; KR18a; Nik+18]. Frank et al. [FAH20] (EthBMC) leverage bounded model checking based on symbolic execution and blockchain state imports to identify this class of faults. When analyzing contracts deployed on the main Ethereum blockchain, it is important to import their state into the analysis tool to avoid spurious false alarms. To compare with EthBMC, we obtained a list of 2856 contracts vulnerable according to EthBMC. Note that only 16 contracts of this set have public ABI information available, making EF/CF rely solely on coverage-guided grey-box fuzzing to identify valid transaction inputs for the remaining contracts. We export the internal state of the contracts at block number 9 069 000 from a *go-ethereum* archive node and import the state into EF/CF and fuzz all contracts until the bug is discovered or for a maximum of 20 min. In total, EF/CF detects 2825 out of 2856 contracts as vulnerable after fuzzing for an average of 28.5 s ($\sigma = 111.4$) until the first bug is discovered, which

demonstrates the high effectiveness of EF↯CF. For 18 contracts, EF↯CF detected no bug in our first run. The remaining contracts had issues due to errors during state export. In a second run with an earlier blockchain state, we find that EF↯CF identifies another 4 vulnerable contracts. After manual analysis, we concluded that 5 of the contracts are only included in the EthBMC dataset due to errors in the EthBMC evaluation. The remaining 12 contracts contain bugs missed by EF↯CF. We argue that this is due to inefficiency when fuzzing without ABI information.

We also applied EF↯CF on 10 356 contracts which EthBMC was unable to analyze due to a timeout after 30 min. This dataset represents contracts that cannot be analyzed with bounded model checking, likely due to problems such as state explosion. In contrast, EF↯CF successfully processes all these contracts and achieves an average code coverage of 72.4 % ($\sigma = 17.9$). Furthermore, EF↯CF detected 85 vulnerable contracts in this set, among which we manually checked 18 contracts with verified source code and found 7 new vulnerabilities. For the remaining 11 contracts, EF↯CF correctly identifies a transaction sequence to gain Ether. However, after manual analysis, we conclude that these transaction sequences allow for legitimate Ether gains and, as such, are false alarms with EF↯CF.

Common False Alarms with EF↯CF During our evaluation of the EthBMC dataset, we identified a number of patterns in smart contracts that lead to false alarms in EF↯CF.

- *Gambling contracts:* EF↯CF identifies the right blockchain state and transactions, such that the attacker always wins. In a real-world deployment, the attacker only has limited control over the environment, such as the blockhash or block number. As such, the attacks that EF↯CF identifies are often not practical on the live blockchain.

- *Airdrops:* Many token contracts implement a so-called *Airdrop* mechanism, where the contract gives out a certain number of free Tokens while the Airdrop is active. EF↯CF deterministically triggers the airdrop to gain tokens. If EF↯CF can sell the tokens again, EF↯CF will report a false alarm.

- *Interest pay-out:* We encountered multiple smart contracts that implement interest payout. For example, the contract allows anyone to invest Ether and then pays out a fixed percentage in interest. EF↯CF will flags this as a vulnerability since it results in Ether gain. These contracts assume that the value of Ether is steadily increasing. However, EF↯CF does not account for any potential changes outside of the blockchain environment, such as the exchange rate between USD and Ether.

We point out that in all these scenarios, there is a way to gain Ether from those contracts, which EF/CF correctly reports. Fundamentally, these contracts violate the basic assumption of EF↯CF's bug oracle that an unrelated third party should not be able to gain Ether by interacting with a contract. Such contracts are a challenge for all analysis tools that attempt to identify bugs in Ether handling (e.g., teEther, EthBMC, etc.).

5.5.2 Reentrancy Vulnerabilities

We evaluated EF⚡CF using a set of contracts vulnerable to reentrancy according to prior studies [Bos+22; Cec+21; Rod+19; Zho+20]. Furthermore, we also compare EF⚡CF with Confuzzius [Tor+21a] and the static source code analyzer Slither [FGG19] (see Table 5.5 for a summary). Confuzzius and Slither feature a more heuristic detection of reentrancy issues than EF⚡CF. Slither defines any state update after an external call as a potential reentrancy bug. Similarly, Confuzzius defines a reentrancy bug as an external call where some state variable is read before the call and written after the call. Note that Confuzzius does not actually generate any reentrant transaction sequence. As many contracts in the set of prior studies [Rod+19; Zho+20] include false alarms, we filter out wrongly detected reentrancy bugs by manually analyzing the reported contracts. Furthermore, many cases are trivial reentrancy bugs, which we summarize as *Trivial-RE* in Table 5.5, which includes reentrancy honeypot contracts [TSS19].

EF⚡CF is highly effective in discovering all known reentrancy issues. However, the prototype implementation of EF⚡CF does not yet support contract creation at runtime. We only noticed dynamic contract creation for "the DAO", which is why we cannot process this contract. The *HODLWallet* contract requires special attention: while this contract is vulnerable to a reentrancy attack, it cannot be exploited to gain Ether. According to our analysis, this contract allows users to invest Ether into the contract, but the contract never returns all the invested Ether. However, when exploiting the contract's reentrancy bug, a user can withdraw all previously invested Ether, diverging from the contract's malicious—but intended—functionality. EF⚡CF's bug oracle does not identify this as a reentrancy bug since no Ether can actually be gained, although EF⚡CF generates a test case containing the necessary reentrant transactions. For the *InstaDice* contract, Confuzzius reports many additional reentrancy issues even when the contract calls into other trusted contracts instead. EF⚡CF, on the other hand, executes also trusted contracts exported from the Ethereum node and produces a full working exploit for the reentrancy bug without reporting false alarms. Remarkably, EF⚡CF is the *only* dynamic analysis tool that is able to accurately identify real-world reentrancy issues, such as the reentrancy bugs in the SpankChain and DSEthToken contracts.

Table 5.5: Results for reentrancy issues for various analysis tools: False Alarms (~), True Alarms (✓), not applicable/incompatible (N/A), or as Missed Bug (×).

Contract	EF⚡CF	Confuzzius	Slither
Example Figure 5.2	✓	~	~
SpankChain [SpCLC]	✓	×	✓
DSEthToken [Rod+19]	✓	✓	N/A
TheDAO [Rod+19]	N/A	✓	N/A
HODLWallet [HODLW; Zho+20]	×	✓	✓
SysEscrow [SYES; Zho+20]	✓	×	✓
InstaDice [INSD; Zho+20]	✓	~	✓
Trivial-RE [TSS19]	✓	✓	✓

Fuzzing Honeypot Reentrancy Torres et al. [TSS19] discussed the phenomenon of honeypot contracts. These contracts are deployed with source code often available on etherscan.io and appear to be vulnerable to, e.g., reentrancy attacks. However, these contracts are, in fact, a scam that targets malicious actors searching for easily exploitable contracts on the blockchain. They require the attacker first to invest a certain amount of Ether to later exploit the seemingly vulnerable contract. However, furtively, the code hides a mechanism that prevents exploitation, locking the previously invested Ether of the attacker. Most of the known reentrancy honeypot contracts use a call to an external library-like contract to revert attack transactions. The deception works by suggesting the source code on etherscan also provides the source code for the external contract when in fact, a different contract is used.

Many of the known honeypot contracts feature very obvious reentrancy vulnerabilities, as these contracts are designed to be easily analyzable (i.e., to lure more attackers). Many of those reentrancy honeypot contracts ended up in various datasets of prior studies [Bos+22; Dur+20]. In the curated version of the smartbugs dataset, the majority of the contracts identified as vulnerable to reentrancy are, in fact, honeypot contracts. This dataset contains 19 reentrancy honeypots and 12 other contracts vulnerable to reentrancy.

These honeypot contracts introduce significant bias into datasets. For example, a tool that detects all honeypot contracts in the curated smartbugs dataset already seems to detect the majority of reentrancy bugs. However, in reality, all these reentrancy bugs follow the exact same code pattern. For this reason, we chose to summarize all these cases as *trivial reentrancy* in Section 5.5.

The reentrancy honeypots can be analyzed in two ways: by relying on the source code only and by importing code and state directly from the blockchain. It is important to distinguish both cases since the contract is exploitable in the former, while in the latter, it is not. We verified that EF↯CF correctly identifies the reentrancy attacks in the first case. Here we deploy a fresh instance of the contract, and the mechanism to prevent exploitation does not work. As such, EF↯CF can correctly identify the reentrancy vulnerability. However, if we export the contract's state from the blockchain—including the external contract that is called—then the mechanism to prevent exploitation is working. In this case, EF↯CF also executes the second external contract, reverting the transaction before the reentrancy takes place. As such, EF↯CF correctly does not report any false alarms here.

Comparison with Sailfish As part of the evaluation of the Sailfish tool, Bose et al. released a list of contracts where reentrancy causes inconsistent state according to Sailfish. This list contains 1904 contracts, of which the Bose et al. verified 26 contracts to be true positives[4]. Among the list of 1904 contracts, EF↯CF identifies vulnerabilities in only 67 contracts. However, in 8 of these 67 contracts, EF↯CF discovers a vulnerability unrelated to reentrancy, e.g., a controlled *delegatecall* vulnerability.

Furthermore, we analyzed the list of verified true positives in more detail. Among the vulnerable contracts reported by the Sailfish tool, EF↯CF correctly identifies 5 contracts that can be exploited with reentrancy to steal Ether. Among these five

[4] `https://github.com/ucsb-seclab/sailfish`

contracts, one is a test contract, one is a known honeypot, and the remaining three contracts are identical. Furthermore, EF/CF identifies one contract that can be exploited due to an access control bug, not a reentrancy bug. As discussed previously, EF/CF identifies the honeypot only as vulnerable when deploying from source code but not when using exported state from the blockchain. We manually identified one contract that seems to be vulnerable to reentrancy, but no Ether is at stake. The remaining contracts exhibit reentrancy patterns, which are likely not exploitable.

For example, the *CommonWallet* contract, depicted in Figure 5.16, is affected by a similar Token-related reentrancy bug, as the *Uniswap-V2* contract [Tor+21b]. Here, the attacker must supply a *ERC777* Token where a *ERC20* Token is expected. Most (legitimate) *ERC20* Token contracts do not perform callbacks to the attacker, and as such, the attacker cannot trigger a reentrancy situation. However, this is different for *ERC777* contracts that feature callbacks by design. This allows the attacker to reenter the *CommonWallet* contract and trigger a reentrancy bug. Currently, EF/CF does not detect Token-related bugs since EF/CF has no concept of Tokens and, as such, does not regard Token-gains as a bug.

However, in reality, this reentrancy bug cannot be used to cause damage. The reason for this is that the attacker would have to cause an integer underflow, which is prevented due to the integer checking leveraged by the contract. As such, EF/CF would not identify a possible reentrancy attack, even if there were a bug oracle for tokens, because there is no way to exploit the reentrancy attack. We verified this by testing EF/CF with a contract that exhibits the same vulnerability but with Ether instead of Tokens. We believe the reason for this false alarm is that Sailfish does not accurately model the transaction-based execution model of the EVM. It will eagerly report a bug without considering that the transaction will revert later, a flaw common to many analysis tools [SSM20].

```
function safeSub(uint256 _x, uint256 _y) internal pure returns (uint256) {
  assert(_x >= _y);
  return _x - _y;
}
function sendTokenTo(address tokenAddr, address to_, uint256 amount) {
  require(tokenBalance[tokenAddr][msg.sender] >= amount);
  /* external call - might cause reentrancy */
  if(ERC20Token(tokenAddr).transfer(to_, amount)) {
    /* state update with underflow check in safeSub, which essentially checks the
       balance again, i.e., */
    //require(tokenBalance[tokenAddr][msg.sender] >= amount);
    tokenBalance[tokenAddr][msg.sender] =
        safeSub(tokenBalance[tokenAddr][msg.sender], amount);
  }
}
```

Figure 5.16: *CommonWallet* reentrancy, which is not exploitable due to the integer overflow check.

Compositional Security We follow the four examples used in the evaluation of the Serif static analyzer [Cec+21] to show the feasibility of detecting compositional security violations with EF/CF. We adapted the contracts from Serif's evaluation set to be deployable and exploitable in a realistic setting. Where Serif relies on manual annotations to detect potential problems, we augment the contracts with assertions that are picked up by EF/CF to detect a potential vulnerability beyond Ether-stealing. EF/CF accurately generates an Ether-stealing reentrancy attack for the Uniswap and Multi-DAO contracts in the Serif dataset. Furthermore, EF/CF generates transactions that violate the assertions for the KV-Store and TownCrier contracts using reentrant transactions.

To evaluate EF/CF's capabilities on a real-world example, we also fuzz the composition of Uniswap V1 and IMBTC, which was attacked with reentrancy [Tor+21b]. We supply EF/CF with the addresses of three contracts that should receive transactions (Uniswap, IMBTC, and the ERC1820Registry) and export the state of these contracts at block number 9 600 000 shortly before the first known attack. We then fuzz this composition on 40 cores for a maximum of 48 h. On average over 10 independent runs it takes EF/CF 1 h and 49 min ($\sigma = 6$ h55 min) to generate a reentrancy exploit that, (1) registers an attacker contract in the ERC1820Registry contract with the right hash to allow ERC777 callbacks, (2) buys IMBTC tokens via the Uniswap contract, and (3) finally exploits the reentrancy to sell them again with a profit.

5.5.3 Problems with Existing Datasets

By now, there are multiple attempts to create standardized datasets to evaluate smart contract analysis tools [Dur+20; GP20]. However, they lack diversity of bugs and especially so with respect to reentrancy bugs. They often only contain the simple same-function reentrancy pattern and contain many honeypot contracts [TSS19].

Ghaleb et al. [GP20] attempt to synthesize a benchmark dataset using bug injection. Unfortunately, we find that they are not suitable to test reentrancy detection based on fuzzing or symbolic execution. The injected reentrancy bugs focus on classical same-function reentrancy and do not cover more complex reentrancy patterns, such as cross-function reentrancy [Rod+19]. Furthermore, the injected patterns are themselves suboptimal. For example, many of the injected reentrancy bugs do not *require* a reentrant call to be exploitable. Figure 5.17 shows an injected bug that is prone to reentrancy. However, the function can also be identified as vulnerable without resorting to a reentrancy attack. For example, MAIAN [Nik+18] detects an Ether leaking vulnerability in this function because the injected function will unconditionally send Ether (line 4) the first time a caller calls the function. Reentrancy is only necessary to repeatedly leak Ether. A second type of injected reentrancy bug is shown in Figure 5.18. This reentrancy bug can never be triggered during fuzzing or symbolic execution. The injected vulnerable function contains a guard, the *require* statement in line 6, that cannot be satisfied. The *require* accesses a storage variable that is never modified. As such, the reentrant call is essentially dead code, assuming the internal state of the contract can be changed arbitrarily. Assuming this ability would be a huge overapproximation and result in a lot of false alarms [FAH20; WC20]. As such, we conclude that the reentrancy bugs injected as part of the *SolidiFI* project are

```
bool not_called_re_ent27 = true;
function bug_re_ent27() public {
  require(not_called_re_ent27);
  if( ! (msg.sender.send(1 ether) ) ){
    revert();
  }
  not_called_re_ent27 = false;
}
```

Figure 5.17: An injected reentrancy bug, which is also detected by bug oracles that only identify leaking ether without considering reentrancy.

```
// initialized with 0
mapping(address => uint) redeemableEther_re_ent25;
function claimReward_re_ent25() public {
  /// since the balances mapping is never written anywhere else,
  /// this require cannot be bypassed by a dynamic analysis tool.
  require(redeemableEther_re_ent25[msg.sender] > 0);
  /// unreachable code
  uint transferValue_re_ent25 = redeemableEther_re_ent25[msg.sender];
  msg.sender.transfer(transferValue_re_ent25);   //bug
  redeemableEther_re_ent25[msg.sender] = 0;
}
```

Figure 5.18: An injected reentrancy bug that cannot be triggered due a guarding condition that cannot be satisfied.

not suitable to test dynamic analysis tools, i.e., fuzzers or symbolic execution-based analyzers.

5.6 Discussion and Related Work

As already discussed in Section 5.1, analyzing smart contract code has been the topic of a rather large body of recent research. Inspired by the success in identifying vulnerabilities in classical software, symbolic execution [Bos+22; FAH20; KR18a; Luu+16; Mos+19; Nik+18] and fuzzing [Cho+21; Gri+20; He+19; Ngu+20; Tor+21a; WC20] have been applied to smart contracts. However, when applying fuzzing and symbolic execution to smart contracts a number of peculiarities have to be considered. Unfortunately, many of the existing fuzzers and symbolic executors do not properly discuss many design decisions. In the remainder of this section, we discuss several design choices of EF/CF and compare them to related work.

Fuzzing Structured Input On the protocol level, smart contracts take a variably-sized byte string as input. However, this byte string is a highly structured input encoded according to the *ABI* definition. If a contract receives wrongly structured input, it will quickly stop execution. Symbolic execution tools handle this by providing

unconstrained symbolic input bytes and querying the SMT solver for solutions. Most current smart contract fuzzers [Gri+20; He+19; Ngu+20] adapt an approach akin to grammar-fuzzing [Bur67; Yan+11] to randomly generate structurally valid inputs based on the *ABI*. While EF↯CF also utilizes the *ABI* to efficiently mutate transaction inputs, it does not solely rely on the *ABI* for mutations. EF↯CF also features raw input mutations and also leverages the lightweight mutations of its base fuzzer. Using the coverage feedback, EF↯CF can discover structurally valid inputs even without the ABI. This approach has two advantages. First, it allows EF↯CF to fuzz contracts where no source code or ABI is available. For example, our evaluation of the EthBMC dataset shows that EF↯CF can also identify access control bugs without any *ABI* definition. Second, EF↯CF can also discover bugs in unusual settings, e.g., EF↯CF can identify bugs that can only be triggered with structurally invalid inputs. EF↯CF can also fuzz inputs that contain byte strings that are further decoded elsewhere (e.g., because the contract forwards all unknown calls to a library contract). Furthermore, EF↯CF can fuzz the return data of external calls, which is ABI-encoded but not part of the ABI definition. However, fuzzing without ABI information is currently less well optimized in EF↯CF (see benchmarks in Section 5.4.2). To improve fuzzing efficiency, static analyses can be used to recover the ABI [Che+21; Gre+19] from the bytecode. Note that due to EF↯CF's probabilistic usage of the smart contract's ABI, EF↯CF does not require a complete or accurate ABI definition. A partial ABI definition covering most of the contract's function would be enough to increase fuzzing efficiency. For example, if the parameters of a function in the ABI cannot be recovered, a generic byte string argument can be used instead, which is then mutated by the regular mutations of EF↯CF.

Bug Oracles Bug oracles are an essential part of an analysis tool. Contrary to native code fuzzers, which feature de-facto standardized bug oracles [Ser+12], there is a wide spectrum of bug oracles in smart contracts. All analysis tools define their own bug oracles, sometimes with slightly different definitions of the same bug classes, making it hard to compare analysis tools. Oyente [Luu+16], OSIRIS [FSS18], Confuzzius [Tor+21a], ILF [He+19], and Sailfish [Bos+22] feature bug oracles that indicate issues, but not necessarily a security vulnerability (or even a bug for that matter). For example, detecting a dependency on the block timestamp might be a sign of a bad attempt at using randomness, or it might be a legitimate use to implement a time-limited sale. Naturally, such oracles also result in a larger number of alarms that might not be security-critical. However, the advantage of such oracles is that they can uncover a larger set of issues. Developers can use these findings to increase code quality by avoiding dangerous code patterns. Another type of analysis tool utilizes Ether loss as a bug oracle to implement exploit generation [FAH20; KR18a; Nik+18] with a very low number of false alarms. EF↯CF utilizes the same bug oracle strategy but also covers reentrancy attacks and complex interactions. The disadvantage of using an Ether-based bug oracle is that Token-related vulnerabilities cannot be easily detected. However, similar to the industry fuzzer Echidna [Gri+20], EF↯CF supports using properties and assertions as bug oracles during fuzzing. This allows EF↯CF to identify various logic bugs, including Token-related vulnerabilities, with the help of developer annotations.

Simulating Interactions All transaction-oriented analysis tools, such as fuzzers and symbolic executors, simulate one or multiple attacker accounts interacting with the target contract. However, not all tools can simulate the full spectrum of possible interactions. More importantly, the majority of existing analysis tools [FAH20; Gri+20; He+19; KR18a; Nik+18; Tor+21a] focus on generating a set of transactions originating from an externally owned account (EOA), which does not allow reentrancy or external calls that return data. Symbolic execution and hybrid fuzzing tools [FAH20; KR18a; Nik+18; Tor+21a] simulate external calls to a certain degree by introducing fresh symbolic values for the returned data. However, reentrancy is generally challenging for symbolic execution tools since it might introduce path explosion [Kal+18]. Most fuzzers refrain from simulating external calls with return values [Gri+20; He+19; Ngu+20]. To detect reentrancy, *sFuzz* introduces dedicated attacker contracts that can be called during fuzzing [Ngu+20]. This allows *sFuzz* to detect certain trivial reentrancy patterns but fails to adapt to contracts requiring more complex interactions (e.g., cross-function reentrancy [Rod+19]). In contrast, in EF↯CF, we assume that transactions originate from an attacker smart contract, allowing more complex interactions such as callbacks with arbitrary return data and reentrancy. This design decision allows EF↯CF to generate reentrancy attacks using a simple Ether-based bug oracle.

Simulating Benign Interactions An important property of a fuzzer is whether it simulates the behavior of benign users, i.e., whether the fuzzer can generate transaction sequences in the form $(t_u, t_a, t'_u, t'_a, \ldots)$, where t_u and t'_u are from a benign user, and t_a and t'_a are from an attacker. For instance, many smart contracts implement the *Owned* pattern: there is one Ethereum account that has special privileges in the contract. This is often used to implement deprecating or upgrading a contract (see Chapter 7), which entails being able to drain all funds from the contract. If the fuzzer aims to reach optimal code coverage, then simulating arbitrary addresses is beneficial to also reach code paths guarded by access control checks. As such, many existing fuzzers, such as ILF [He+19] and Confuzzius [Tor+21a] adopt this behavior by default. However, while this approach leads to good code coverage, it also entails a larger number of false alarms. For example, in contracts with transferable ownership, the fuzzer will make the simulated owner transfer the ownership to an attacker account. In turn, this would allow the attacker to drain the funds of the contract. Obviously, this is a false alarm since the real owner would never transfer ownership to an attacker. However, this ownership concept is widespread in smart contracts, making existing fuzzers that adopt this behavior produce many false alarms. For example, He et al. [He+19] describe a case study of a vulnerability in the *Grid* contract. However, this is a false alarm as it would require interaction with the contract by both the contract owner and a benign user. Both need to set specific contract parameters such that the attacker is able to gain Ether. From our analysis of this contract, we concluded that this is not something that the owner would do. Note that EF↯CF raises this exact same false alarm when we enable simulating transactions originating from the owner. Ideally, one would simulate only benign user interactions. Unfortunately, deciding which actions a benign user would perform is, in general, not possible since this depends on the specific semantics and intentions behind a smart contract. As such, we opted to disable the simulation of any non-attacker-controlled accounts in EF↯CF.

Testing Multi-Contract Setups Smart contracts increasingly feature dependencies on third-party contracts. To support such contracts, EF↯CF allows the contract under test to call other trusted contracts. However, EF↯CF, as almost all other analysis tools, will only execute transactions that target the smart contract in question. Hence, EF↯CF will not discover bugs that can only be triggered by an unsuspected state change in the trusted dependencies of the contract under test. For example, certain attacks require the attacker to first acquire tokens before proceeding to attack another contract that handles the token. Recently, Echidna [Gri+20] introduced a *multi-abi* mode, where the fuzzer is allowed to call functions on multiple smart contracts. Recent incidents such as the attacks against the CREAM Finance [CREAM] or the Revest [REVST] smart contracts show that multi-contract analysis is required for the automatic analysis of complex DeFI applications.

In this paper, we show that EF↯CF can handle even complex inputs and dependencies between transactions. We already extended EF↯CF with an experimental mode that shows that EF↯CF can also be extended to multi-target fuzzing. This extension is conceptually quite simple, as the input format just requires a selector field for the target of a transaction that is then consumed by the harnessing code to select the target contract for a transaction. However, this feature also requires careful performance engineering to make multi-target fuzzing efficient while simultaneously avoiding a decrease in single-target fuzzing performance. Additionally, when fuzzing targets that are exported from the blockchain, such a mode requires additional analysis to determine the set of contracts that are useful to receive fuzzed transactions. As such, we leave extending EF↯CF with such a mode as an interesting direction for future work.

5.7 Conclusion

Due to their high popularity, smart contracts are increasingly used to encode complex business logic in Ethereum and are becoming prominent features of other blockchain platforms. Hence, there is a high demand for developing highly efficient and scalable techniques to perform automated security analysis of smart contracts.

Unlike existing fuzz testing approaches for Ethereum smart contracts, our framework EF↯CF specializes in high-throughput fuzzing. This is achieved by employing two techniques to speed up fuzzing: (a) a novel translation technique that translates contract bytecode to native C++ code and (b) with efficient structural mutations on transaction sequences and the associated transaction inputs. EF↯CF offers a variety of advantages compared to existing testing approaches, including increased code coverage, reduced time to discover bugs, the capability to model complex interactions (e.g., reentrant transactions), and analysis capabilities for highly complex contracts. Given the popularity of cryptocurrencies along with the steadily increasing value of assets that smart contracts manage, it is important to improve the state of automated security testing of smart contracts. As such, we will release EF↯CF and the benchmarks that we developed for our evaluation as open-source software. We hope this allows developers to leverage efficient automated testing of contracts and fosters additional research in smart contract security.

CHAPTER 6

MITIGATION OF REENTRANCY ATTACKS

In previous chapters, we discuss vulnerabilities in smart contracts (Section 3.2) and how to automatically identify such vulnerabilities (Chapter 5). However, this still leaves already deployed smart contracts at risk. We note that fixing discovered bugs in smart contracts is particularly challenging due to three key challenges: (1) the code of a smart contract immutable after deployment, (2) smart contract owners are anonymous, i.e., responsible disclosure is usually infeasible, (3) existing approaches are mostly performing offline analysis and are susceptible to missing unknown run-time attack patterns. As a consequence of the immutability, approaches that prove correctness or absence of a certain type of vulnerability [Kal+18; Luu+16; Sch+20b; Tsa+18] are only important for the development of future smart contracts, but leave already deployed (legacy) contracts vulnerable. More specifically, to deal with a vulnerable contract and restore a safe state, the owner of the contract must deprecate the vulnerable contract, move all funds out of the contract, deploy a new contract, and move the funds to the new contract. This process is largely cumbersome since the address of the vulnerable contract might be referenced by other contracts. We discuss how to streamline the process of updating smart contracts in Chapter 7. Finally, offline analysis techniques typically cannot fully cover the run-time behavior of a smart contract, thereby missing new attack patterns exploiting code constructs that were believed to be not exploitable.

In this chapter, we show how to tackle these challenges and how to prevent legacy, potentially vulnerable, smart contracts from being exploited (1) without changing the smart contract code, and (2) without possessing any semantic knowledge of the smart contract. We focus our analysis on reentrancy attacks, which are arguably the most challenging vulnerability class. Reentrancy issues continue to be exploited in multiple major incidents since the infamous "The DAO" incident [REVST; CREAM; Jen16; Tor+21b].

Contributions In this chapter, we present the design and implementation of Sereum (Secure Ethereum), which is able to protect existing, deployed contracts against reentrancy attacks by performing run-time monitoring of smart contract execution. Given our run-time monitoring technique, Sereum is able to cover the actual execution

flow of a smart contract to accurately detect and prevent attacks. As such, our approach also sheds important light on the general problem of the incompleteness of any offline static analysis tool. To underline this fact, we present new reentrancy attack patterns (Section 6.2), which are not covered accurately in static analysis tools.

We present our prototype implementation of Sereum for the EVM, in which we introduce a hardened EVM that leverages taint tracking to monitor the execution of smart contracts. While taint tracking is a well-known technique to detect leakage of private data [Enc+14] or memory corruption attacks [CLO07a], we apply it for the first time to a smart contract execution platform. Specifically, we monitor data flows from storage variables to control-flow decisions. Our main idea is to automatically introduce write locks (i.e., mutexes), which prevent the contract from updating storage variables in other invocations of the same contract in the same transaction. Sereum prevents any write to variables, which would render the contract's state inconsistent with a different re-entered execution of the same contract. Sereum also rolls back transactions that trigger an invalid write to variables—thereby effectively preventing reentrancy attacks. Furthermore, Sereum can also be used as a passive detection tool, where it does not perform rollbacks of attack transactions but only issues a warning for detected attacks.

We perform an extensive evaluation of our approach by re-executing a large subset of transactions of the Ethereum blockchain (Section 6.5). Our results show that Sereum detects all malicious transactions related to the DAO attack and only incurs 9.6 % run-time overhead. We further verify our findings by using existing vulnerability detection tools and manual code analysis on selected contracts. Although Sereum exhibits a false alarm rate of only 0.06 %, we provide a thorough investigation of false alarms associated with our approach and other existing static analysis tools [Luu+16; Tsa+18]. We thereby demonstrate that Sereum provides improved detection of reentrancy attacks compared to existing approaches with negligible run-time overhead.

The basis for this chapter is the following publication:
"Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks". ***26th Annual Network and Distributed System Security Symposium (NDSS)*, 2019**. Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi

6.1 Problem Statement

In this chapter, we propose a defense that protects existing, deployed smart contracts against reentrancy attacks in a backward-compatible way without requiring source code or any modification of the contract code. As mentioned earlier, reentrancy patterns are prevalent in smart contracts and require developers to carefully follow the implementation guidelines [SolWd].

We observed in various incidents [REVST; CREAM; Jen16; Tor+21b] that contracts that are vulnerable to reentrancy attacks often suffer from high losses of tokens or Ether. Prior to our study, the only publicly documented reentrancy attack,was against the "The DAO" contract [Pri16]. Our evaluation shows that reentrancy attacks have not yet been launched against other contracts (except for some new minor incidents we will describe in Section 6.5). However, since our original study, various incidents due

to reentrancy attacks have been demonstrated [REVST; CREAM; Tor+21b]. These findings demonstrate that a systematic defense against reentrancy attacks is required to protect these contracts from being exploited.

As discussed in Chapter 5, the majority of defenses deploy static analysis and symbolic execution techniques to identify reentrancy vulnerabilities. While these tools surely help avoid reentrancy for new contracts, how to protect existing ones remains open. That is, fixing smart contract vulnerabilities in deployed smart contracts using existing analysis tools is highly challenging owing to the immutability of smart contract code and the anonymity of smart contract owners. We discuss smart contract upgrade strategies in more detail in Chapter 7.

Apart from these fundamental limitations, we also observe that existing detection approaches fail to effectively detect all reentrancy vulnerabilities or suffer from a high number of false positives. More specifically, we note that existing approaches can be undermined by advanced reentrancy attacks. To this end, we systematize existing reentrancy patterns and identify new reentrancy patterns. We show that existing tools do not flag as reentrancy vulnerabilities but are nevertheless exploitable. We classify reentrancy attacks into (1) cross-function reentrancy, (2) delegated reentrancy, (3) create-based reentrancy, and (4) unconditional reentrancy. We describe the patterns in detail in Section 6.2. While cross-function reentrancy vulnerabilities have been partially discussed in the Ethereum community [Conb; Dai16b], we believe that our study is the first presentation of delegated and create-based reentrancy attacks. All of these attacks are either missed or imprecisely detected by the state-of-the-art detection tools such as Oyente [Luu+16], Securify [Tsa+18], and ZEUS [Kal+18]. In what follows, we present attacks that exploit these reentrancy patterns and discuss why existing tools cannot accurately mark the contract code as vulnerable. As we show, these attacks map to standard programming patterns and are highly likely to be included in existing contracts. To re-produce our attacks and test them against the public detection tools, the source code of the vulnerable contracts and the corresponding attacks is available online [Rod].

6.2 New Reentrancy Attack Patterns

This section presents our systematization of reentrancy patterns. We base the systematization presented in this section on our previous publication [Rod+19], where we described several reentrancy patterns for the first time in academic literature. Note that we extended the original systematization by the *unconditional reentrancy* pattern.

Same-Function Reentrancy

Same-function reentrancy became publicly known due to the infamous "The DAO" incident and is arguably the most widely known type of reentrancy. As a consequence, many analysis tools are also designed to detect the same-function reentrancy pattern [Kal+18; Luu+16; Tsa+18]. The same-function reentrancy pattern is arguably rather simple: the attacker forces a call to the same function in a reentrant manner. Since the function is not reentrancy-safe, the attacker can gain some advantage, e.g.,

stealing Ether. Figure 3.6 shows an example of a same-function reentrancy vulnerability. Here the attacker can repeatedly call the *withdraw* function to bypass the balance check.

Cross-Function Reentrancy

```
1  mapping (address => uint) tokenBalance;
2  mapping (address => uint) etherBalance;
3
4  function withdrawAll() public {
5    uint etherAmount = etherBalance[msg.sender];
6    uint tokenAmount = tokenBalance[msg.sender];
7    if (etherAmount > 0 && tokenAmount > 0) {
8      uint e = etherAmount + (tokenAmount * currentRate);
9      etherBalance[msg.sender] = 0;
10     // cannot re-enter withdrawAll()
11     // However, can re-enter transfer()
12     msg.sender.call.value(e)();
13     // state update causing inconsistent state
14     tokenBalance[msg.sender] = 0;
15   }
16 }
17 function transfer(address to, uint amount) public {
18   // uses inconsistent tokenBalance (>0) when re-entered
19   if (tokenBalance[msg.sender] >= amount) {
20     tokenBalance[to] += amount;
21     tokenBalance[msg.sender] -= amount;
22   }
23 }
```

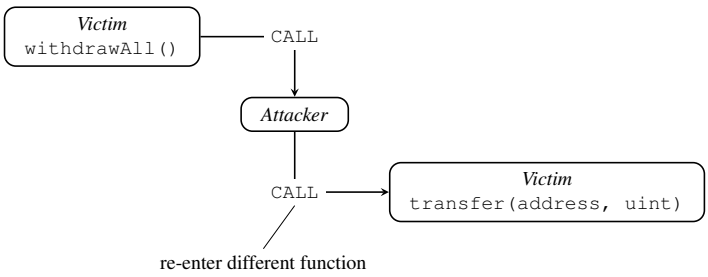


Figure 6.1: The upper part shows the relevant code from a contract, which is modeled after a typical ERC20 Token contract. This code contains a cross-function reentrancy bug. The lower part shows the call chain during the attack. The attacker first calls *withdrawAll* and then reenters the *transferToken* function. For a successful attack, we assume the attacker is then able to exchange the tokens for Ether.

The cross-function reentrancy attack pattern exploits the fact that a reentrancy attack is not limited to a single function. Instead, the attack can also span over multiple functions of the victim contract. This means that instead of calling the same

function again, the attacker instead calls into another public function of the smart contract. Cross-function reentrancy attacks apply to all functions that operate on the same state variables. If two functions do not share access to a state variable, they are independent. As such, the attacker cannot modify the behavior of the smart contract using the reentrant invocation. Our results (see Section 6.5) show that such cross-function reentrancy attacks are equally dangerous as traditional same-function reentrancy. Moreover, more recent real-world incidents were due to cross-function reentrancy attacks, such as the *Uniswap V1* incident [Tor+21b].

Consider the short snippet from a contract, which resembles an ERC20 Token, depicted in Figure 6.1. The function *withdrawAll* performs a state update on the *tokenBalance* variable after an external call. Same-function reentrancy is thwarted in this case because the *etherAmount* is set to zero before the external call. The condition check in line 7 cannot evaluate to true anymore. This effectively prevents an attacker from reentering the *withdrawAll* function. However, the attacker can still call other functions in a reentrant manner. For example, the attacker can reenter the *transfer* function. Note that this function also uses the—now inconsistent—*tokenBalance* variable. As a result, the reentrant call to the *transfer* function allows the attacker to transfer tokens to another address before the *withdrawAll* finishes execution. The attacker utilizes this to move the tokens to another address while calling the *withdrawAll* function, essentially creating new tokens out of thin air.

In general, detecting cross-function reentrancy is challenging for any static analysis tool due to the potential state explosion in case every external call is checked to be safe for every function of the contract. For this reason, many early academic analysis tools do not accurately address cross-function reentrancy. Namely, ZEUS [Kal+18] omits to verify cross-function reentrancy safety, and Oyente [Luu+16], an early symbolic executor, does not flag the code depicted in Figure 6.1 as vulnerable to reentrancy. This is because Oyente reports reentrancy only if the same instruction can be reached in a reentrant call (i.e., by definition, same-function reentrancy). However, recent advances in static analysis methods make it possible to identify cross-functions issues or even prove their absence [Cec+21; Sch+20b]. Similarly, recent work in symbolic execution tools also allows the detection of cross-function reentrancy vulnerabilities. For example, Manticore [Mos+19] is able to detect cross-function reentrancy issues. To avoid state explosion during analysis, many analysis tools utilize the over-approximating *no write after call* policy, which we discussed earlier in Section 3.2.3. This also captures cross-function reentrancy but is overly restrictive in general. In general, ECFChecker [Gro+18] is able to detect cross-function reentrancy attacks. However, during our evaluation, we were able to construct a contract that can be exploited with a cross-function reentrancy attack without being detected by ECFChecker (see Section 6.5).

Delegated Reentrancy

The key issue with the delegated reentrancy patterns is that the attack exploits the fact that the reentrancy is hidden within a `DELEGATECALL` or `CALLCODE` instruction. These EVM instructions are intended for implementing library contracts and allow a contract to execute the code of another contract in its own execution context. In

```
1  library Lib {  // Library contract
2    function send(address to, uint256 amount) public {
3      to.call.value(amount)(); // CALL
4    // ...
5  }
6  contract Victim {
7    mapping (address => uint) public credit;
8    Lib lib; // address of library contract
9    // ...
10   function withdraw(uint amount) public {
11     if (credit[msg.sender] >= amount) {
12       // DELEGATECALL into Library
13       address(lib).delegatecall(
14         abi.encodeWithSignature("send(address,uint256)", to, amount));
15       // state update after DELEGATECALL
16       credit[msg.sender] -= amount;
17     }
18 // ...
```

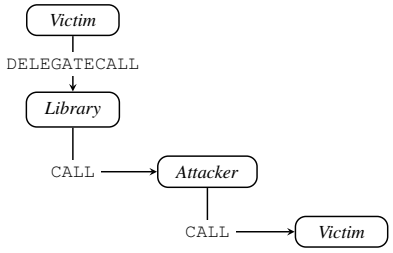


Figure 6.2: The upper part shows the relevant solidity source code. The lower part shows the call chain for a delegated reentrancy attack. Analyzed in isolation, the *Victim* and the *Library* contract are not vulnerable to reentrancy. However, when the *Victim* contract is combined with the *Library* contract, it becomes vulnerable. In this simplified case the *Library* contract is simply used for sending Ether.

Ethereum, library contracts are simply other contracts deployed on the blockchain. When a contract invokes a library contract with one of the special call instructions, they share the same execution context. As such, a library has full control over the calling contract's funds and internal state (storage region). As with traditional software, libraries can be used to avoid duplicating code, as every library use re-uses the same code, which is deployed only once on the blockchain. Furthermore, it also allows a contract to update functionality by switching to a newer version of the library (see Chapter 7 for a longer discussion).

The delegated reentrancy pattern always affects a combination of contract and library contract. We consider such a pair to be vulnerable to a delegated reentrancy if the state update takes place in one contract and the external call is issued in the other. For example, the improper state update happens in the library after the contract has already performed the external call. The problem is that when each one

of the contracts is analyzed in isolation, none of the contracts exhibit a reentrancy vulnerability. The reentrancy vulnerability emerges Only when both contracts are considered in combination. Figure 6.2 depicts a simplified example of a contract vulnerable to delegated reentrancy. The main contract uses a library contract for issuing external calls, and as such, the external call is hidden from analysis tools.

Detecting delegated reentrancy attacks is a significant challenge for existing offline analysis tools. The problem is that during analysis, the code of the library contract is not available to the analysis tool. As such, offline analysis tools, such as Oyente [Luu+16] or Securify [Tsa+18], fail to identify the delegated reentrancy vulnerability due to under-approximation. However, other static analyzers are aware of this pattern and do over-approximate and identify such issues [Gie+22]. Fuzzing and symbolic execution techniques could potentially leverage the current blockchain state to infer which library the contract calls. This allows the executor to fetch the code of the library contract and continue symbolic execution. However, there is one downside to this approach: it is not a future-proof solution, as an updated version of the library contract might introduce a new vulnerability in the future. Dynamic analysis tools that analyze transactions at runtime can easily detect such delegated reentrancy issues. For example, dynamic analysis tools, such as Sereum (see Chapter 6) or ECFChecker [Gro+18], are able to detect delegated reentrancy attacks, as they analyze a combination of contracts and libraries.

Create-Based Reentrancy

Similar to the delegated reentrancy pattern, this pattern abuses the fact that a contract's constructor can issue further external calls. Recall that contracts can either be created by accounts (with a special transaction) or by other contracts using the `CREATE` family of opcodes. In solidity, a new contract can be created with the *new* keyword, which translates to the `CREATE` instruction on the EVM level. Whenever a new contract is created, the constructor of that contract will be executed immediately. Usually, the newly created contract will be trusted and, as such, does not pose a threat. However, the newly created contract can issue further calls in the constructor to other—possibly malicious—contracts. To be affected by a create-based reentrancy issue, the victim contract must first create a new contract and then update its own internal state, resulting in a possible inconsistent state. The newly created contract must also issue an external call to an attacker-controlled address. This allows the attacker to reenter the victim contract and exploit the inconsistent state.

Create-based reentrancy is overlooked in many state-of-the-art analysis tools. For example, Oyente, Manticore, and ECFChecker consider only `CALL` instructions when checking for reentrancy vulnerabilities. Hence, they all fail to detect create-based reentrancy attacks. Similar to delegated reentrancy, the create-based reentrancy vulnerability emerges only when two contracts are combined. Thus, the contracts must also be analyzed in combination.

Securify [Tsa+18] and Mythril [Conc] do not consider `CREATE` as an external call and thus do not flag subsequent state updates. Due to their conservative policy of flagging all state updates after external calls, they could simply include the `CREATE` instruction into the list of external call instructions, allowing them to handle create-

based reentrancy. However, this will almost always result in a false alarm. The `CREATE` instruction returns the address of the newly created contract. This address is needed for further interactions with the newly created contract. As such, it is very likely that this address is saved into the storage area of the caller, i.e., a write after an external call.

Unconditional Reentrancy

```
contract Victim {
  mapping (address => uint) private balance;

  function withdrawAll() public {
    msg.sender.call{value: balance[msg.sender]}("");
    balance[msg.sender] = 0;
  }
  // ...
}
```

Figure 6.3: Unconditional reentrancy pattern, where a successful attack does not bypass any condition.

Typically a reentrancy attack will try to subvert a business logic check of an application. Every high-level check (e.g., an if statement, an assert statement, etc.) is implemented as a conditional jump (`JUMPI`) on the EVM level. Essentially, here we have a data-dependency of a conditional control-flow instruction, the `JUMPI`, to an inconsistent value loaded from storage. Most real-world reentrancy attacks observed and all reentrancy patterns discussed so far share this property. However, the root cause of the reentrancy bug is not necessarily a data dependency on a control-flow instruction. Similarly, a data-dependency of *a state-changing instruction* to an inconsistent storage value can cause a reentrancy bug.

One very simple example is the implementation of a *withdrawAll* function depicted in Figure 6.3, which will simply let the caller withdraw all previously invested Ether. In this example, the contract *VulnBank* unconditionally sends ether to the sender (`msg.sender`). It is not necessary to perform any additional checks. If the sender has not invested any Ether, the amount of transferred ether is 0. Since the gas usage must be paid for by the caller of the contract, such an invocation does not harm the contract. However, an attacker can reenter this function and exploit the inconsistent state to drain the contract of Ether.

6.3 Design Overview

In this section, we describe our approach to detect reentrancy attacks based on run-time monitoring at the level of EVM bytecode instructions. Our approach, called Sereum (Secure Ethereum), is based on extending an existing Ethereum client, which we extend to perform run-time monitoring of contract execution.

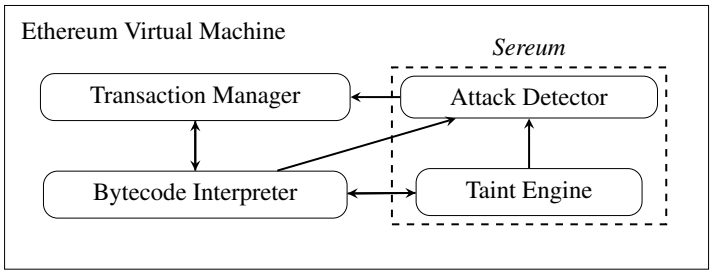


Figure 6.4: Architecture of enhanced EVM with run-time monitoring.

Architecture Figure 6.4 shows an overview of the SEREUM architecture. For a standard Ethereum client, the EVM features a bytecode interpreter, which is responsible for executing the code of the smart contracts. The transaction manager executes, verifies, and commits new and old transactions. Every invocation of a smart contract happens within a transaction in the Ethereum network. Transactions are then grouped together in blocks. Every miner in the Ethereum network must execute all transactions locally to generate the new state of the latest block on the blockchain. To verify the blockchain's current state, a node in the Ethereum network has to replay the execution of all previous transactions.

SEREUM extends the EVM by introducing two new components: (1) a taint engine, and (2) an attack detector. The taint engine performs dynamic taint-tracking. Dynamic taint-tracking, or dynamic data-flow analysis in general, assigns labels to data at predefined sources and then observes how the labeled data affects the execution of the program [SAB10]. To the best of our knowledge, SEREUM is the first dynamic taint-tracking solution for smart contracts. The attack detector utilizes the taint engine to recognize suspicious states of program execution, indicating that a reentrancy attack is happening in the current transaction. It interfaces with the transaction manager of the EVM to abort transactions as soon as an attack is detected.

Detecting Inconsistent State To effectively reason about a malicious reentrance into a contract, we need to detect whether a contract acts on inconsistent internal state. Note that any persistent internal state is stored in the storage memory region of the EVM. Variables that are shared between different invocations of a contract are always stored in the *storage* region. As such, only the storage region is relevant for reentrancy detection. Thus, SEREUM applies taint tracking to storage variables as these are the only internal state variables capable of affecting a contract's control flow in a subsequent (reentered) invocation of the contract. The intuition behind SEREUM's approach is that if a control-flow decision is dependent on storage variables, an attacker can manipulate the outcome of a conditional branch decision by reentering the contract. In turn, manipulating a conditional branch allows the attacker to manipulate the overall behavior of the contract. Hence, reentrancy attacks apply to contracts that execute conditional branches dependent on persistent internal state, i.e., the storage region.

The main idea behind SEREUM is to detect state updates, i.e., altering of storage variables, after a contract (denoted as *Victim* contract) calls into another contract (denoted as *Attacker* contract). Notice that not all state updates resemble malicious behavior, but only those where *Victim* is reentered and acts upon the updated state. Typically, the goal of reentrancy attacks is to bypass validity checks in the business logic of the *Victim* contract. As such, SEREUM focuses only on conditional jumps and the data that influences the conditional jumps. Notice that it is also possible for a contract to transfer Ether without performing any validity check. Deploying such a contract would be highly dangerous and inefficient due to the unnecessary consumption of gas. The original version of SEREUM did not explicitly capture such cases. However, in an extended version of SEREUM, we cover this kind of reentrancy attack by issuing write-locks not only for behavior-changing variables but also for variables that are passed to other contracts during external calls (such as Ether amount or call input).

Consider the example shown in Figure 6.5, *Victim* calls into the *Attacker* contract. The *Attacker* then forces reentrancy into the *Victim* contract by calling into the *Victim* again. The second reentered invocation of *Victim* reads from a storage variable and takes a control-flow decision based on that variable. After the *Attacker* contract eventually returns again to *Victim*, the *Victim* contract will update the state. However, at this point, it is clear that the reentered *Victim* used a wrong value read from inconsistent internal state for its conditional branch decision.

The key observation is that inconsistent state arises if (1) a contract executes an external call to another contract, (2) the storage variable causing inconsistency is used during the external call for a control-flow decision or as a parameter to an external call, and (3) the variable is updated after the external call returns. Next, we describe in more detail how the taint engine and the attack detector detect inconsistent state at the EVM level.

The primary mechanism to detect reentrancy here is detecting the influence of an inconsistent storage variable on a control-flow decision, i.e., a conditional jump, during execution. However, while this captures a large variety of vulnerabilities, it does not capture unconditional reentrancy. As such, we also track the usage of inconsistent variables as parameters to external calls. More specifically, we analyze the *call value*, i.e., the transferred cryptocurrency, and the *input data* provided to the called smart contract.

Taint Engine and Attack Detector To detect state updates, which cause inconsistency, we need to know which storage variables the contract used for control-flow decisions. On the EVM bytecode level, a smart contract implements any control-flow decision as a conditional jump instruction. Consequently, we leverage our taint engine to detect any data flow from a storage load to the condition processed by a conditional jump instruction. This ensures that we only monitor those conditional jumps which are influenced by a storage variable. For every execution of a smart contract in a transaction, SEREUM records the set of storage variables, which were used for control-flow decisions. Using this information, SEREUM introduces a set of locks that prohibit further updates for those storage variables. If a previous invocation of the contract

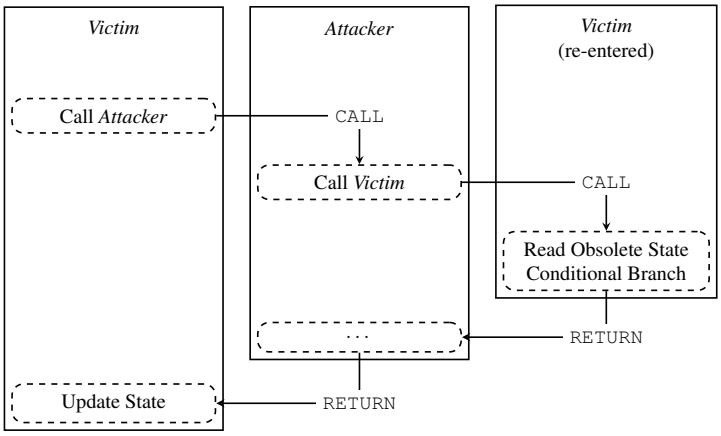


Figure 6.5: reentrancy attack exploits inconsistent state among different invocations of a contract.

attempts to update one of these variables, SEREUM reports a reentrancy problem and aborts the transaction to avoid exploitation of the reentrancy vulnerability.

In the simplest case, the attacker directly reenters the victim contract. However, the attacker might try to obfuscate the re-entrant call by first calling an arbitrarily long chain of nested calls to different attacker-controlled contracts. Furthermore, during the external call, the attacker can reenter the victim contract several times, possibly in different functions (as shown in the cross-function reentrancy attack described in Section 6.2). This has to be taken into account when computing the set of locked storage variables. To tackle these attacks, SEREUM builds a dynamic call tree during the execution of a transaction. Every node in the dynamic call tree represents a call to a contract, and the depth of the node in the tree is equal to the depth of the contract invocation in the call stack of the EVM. We store those storage variables which influence control-flow decisions as set D_i for every node i in the dynamic call tree. The set of storage variables L_i that are locked at node i is the union of D_j for any node j of the same contract as i that belongs to the sub-tree spanning from node i.

Example of Dynamic Call Tree Figure 6.6 depicts an example for SEREUM's generation of a dynamic call tree for a given Ethereum transaction. A possibly malicious contract A reenters a vulnerable contract C multiple times at different entry points (functions). First, as shown on the left sub-tree, contract A calls C, C calls A, and A finally reenters C. This sub-tree would be equivalent to a classical reentrancy attack, as shown previously in Figure 3.6. The variables locked during the first execution of contract C (node marked with 2) are impacted only by the lower nodes in the call tree. The second execution of contract C (in node 4) uses the storage variable V_1 for deciding a conditional control flow. Hence, this variable must not be modified after the call in the execution of node 2.

In contrast, the right side of the call tree contains a more diverse set of nodes. For instance, the right part of the call tree could be part of a cross-function reentrancy attack. We can observe that different functions were called in the various re-entrant

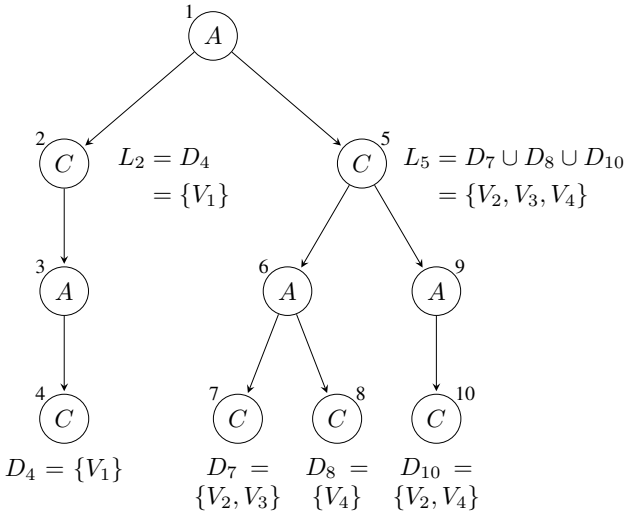


Figure 6.6: Dynamic call tree of an Ethereum transaction. Contract A is reentered several times. V_k are storage variables. D_i is the set of storage variables, which influence control-flow decisions in node i. L_i is the set of storage variables, which are locked at node i and cannot be updated anymore.

invocations of C because the variables used for conditional branches are different. Note that none of the sets D_5, D_7, D_8, and D_{10} are equal. Contract C performs two calls into A in node 5. These calls reenter C in nodes 7, 8, and 10. For the execution of C from node 5, we lock all variables from the subtree below node 5. Note that although variable V_1 is locked in node 2, it is not in the set of locked variables L_5. This means that no further calls starting from node 5 have used the variable V_1 for a control-flow decision; thus, V_1 can be safely updated in node 5, which will not change the behavior in any of the nodes 7, 8 and 10 unexpectedly.

A naive implementation of Sereum could just lock all variables which were used for control-flow decisions. However, as we can see from Figure 6.6, this would result in unnecessary locking of variables when complex transactions are executed. In turn, this would also result in a high number of false alarms. For example, contract C can safely update the state variables V_2, V_3, and V_4 in node 2 because they were not used for conditional branches during the execution of node 4. Similarly, node 5 can safely update V_1 even though it was used for a control-flow decision in a re-entrant call at node 4.

The dynamic call tree allows Sereum to tackle the challenging new reentrancy attacks we discuss in Section 6.2. Recall that detecting *cross-function reentrancy* is challenging for static analysis tools due to potential state explosion. Since Sereum performs dynamic analysis, it does not suffer from such kind of weakness; it only analyzes those cross-function reentrant calls that actually occur at run-time. Similarly, *delegated reentrancy* attacks are detected as Sereum, in contrast to existing tools, does not inspect contracts in isolation but analyzes and monitors exactly the library code which is invoked when a transaction executes. That is, as an extension to the

Ethereum client, Sereum can access the entire blockchain state and hence retrieve the code of every invoked library contract. Our taint engine simply propagates the taints through the library code. This also naturally covers any future updates of the library code. Next, we describe the implementation details of Sereum.

6.4 Implementation Details

We implemented Sereum based on the popular *go-ethereum*[1] project, whose client for the Ethereum network is called *geth*. In our implementation, we extended the existing EVM implementation to include the taint engine and the reentrancy attack detector.

We faced one particular challenge in our implementation: variables stored in the *storage* memory region are represented on the EVM bytecode level as load and store instructions to specific addresses, i.e., any type information is lost during compilation. Hence, only storage addresses are visible on the EVM level. Most storage variables, such as integers, are associated with one address in the storage area. However, other types, such as mapping of arrays, use multiple (not necessarily) adjacent storage addresses. As such, Sereum tracks data-flows and sets the write-locks on the granularity of storage addresses.

In the remainder of this section, we describe how Sereum tracks taints from storage load instructions to conditional branches to detect storage addresses that reference values that affect the contract's control flow. Furthermore, we show how Sereum performs attack detection by building the dynamic call tree and propagating the set of write-locked storage addresses.

Taint Engine

Taint tracking is a popular technique for analyzing data-flows in programs [SAB10]. First, a *taint* is assigned to a value at a pre-defined program point, referred to as the so-called taint source. The taint is propagated throughout the execution of the program, along with the value it was assigned to. Taint sinks are pre-defined points in the program, e.g., certain instructions or function calls. If a tainted value reaches a taint sink, the Sereum taint engine will issue a report and invoke the attack detection module. Taint analysis can be used for both static and dynamic data-flow analysis. Given that we aim to achieve run-time monitoring of smart contracts, we leverage dynamic taint tracking in Sereum.

To do so, we modified the bytecode interpreter of *geth*, ensuring that it is completely transparent to the executed smart contract. Our modified bytecode interpreter maintains shadow memory to store taints separated from the actual data values, which is a common approach for dynamic taint analysis. Sereum allocates shadow memory for the different types of mutable memory in Ethereum smart contracts (see Section 3.2.1). The stack region can be addressed at the granularity of 32-byte words. Thus, every stack slot is associated with one or multiple taints. The storage address space is also accessed at 32-byte word granularity, i.e., the storage can be considered as a large array of 32-byte words, where the storage address is the index into that array. As a

[1] `https://github.com/ethereum/go-ethereum`

result, we treat the storage region similar to the stack and associate one or multiple taints for every 32-byte word. However, unlike the stack and storage address space, the memory region can be accessed at byte granularity. Hence, we associate every byte in the memory address space with one or multiple taints. To reduce the memory overhead incurred by the shadow memory for the memory region, we store taints for ranges of the memory region. For example, if the same taint is assigned to memory addresses 0 to 32, we only store one taint for the whole range. When only the byte at address 16 is assigned a new taint, we split the range and assign the new taint only to the modified byte.

We propagate taints through the computations of a smart contract. As a general taint propagation rule for all instructions, we take the taints of the input parameters and assign them to all output parameters. Since the EVM is a stack machine, all instructions either use the stack to pass parameters or have constant parameters hard-coded in the code of the contract. Hence, for all of the computational instructions, such as arithmetic and logic instructions, the taint engine will pop the taints associated with the instruction's input parameters from the shadow stack, and the output of the instruction is then tainted with the union of all input taints. In contrast, constant parameters are always considered untainted. This ensures that we accurately capture data flows within the computations of the contract. One exception to the general tainting rule is the `SWAP` instruction family, which swaps two items on the stack. The taint engine will also perform an equivalent swap on the shadow stack without changing taint assignments. Similarly, whenever a value is copied from one of the memory areas to another area, we also copy the taint between the different shadow areas. For instance, when a value is copied from the stack to the memory area, i.e., the contract executes a `MSTORE` instruction, the taint engine will pop one taint from the shadow stack and store it in the shadow memory region. The EVM architecture is completely deterministic; smart contracts in the EVM can only access the blockchain state using dedicated instructions. That is, no other form of input or output is possible. This allows us to completely model the data flows of the system by tracking data flows at the EVM instruction level.

For reentrancy detection, as described in Section 6.3, we only need one type of taint called *DependsOnStorage*. The taint source for this taint is the `SLOAD` instruction. Upon encountering this instruction, the taint engine creates a taint, which consists of the taint type and the address passed as an operand to the `SLOAD` instruction. The conditional `JUMPI` instruction is used as a taint sink. Whenever such a conditional jump is executed, the taint engine checks whether the condition value is tainted with a *DependsOnStorage* taint. If this is the case, the taint engine will extract the storage address from the taint and add it to the set of variables that influenced control-flow decisions. Our implementation supports an arbitrary number of different *DependsOnStorage* taints. This allows Sereum to support complex code constructs, e.g., control-flow decisions that depend on multiple storage variables.

Example for Taint Assignment and Propagation Figure 6.7 shows an example for taint propagation in Sereum. The figure depicts a snippet of Ethereum bytecode instructions. In this snippet of instructions, a data flow exists from the `SLOAD`

instruction in line 1 to the conditional jump instruction in line 4. The `SLOAD` instruction will load a value from the storage memory region. The first and only parameter to `SLOAD` is the address in the storage area. The `JUMPI` instruction takes two parameters: the jump destination and the condition of whether the jump is to be performed. Recall that all instruction operands except for the `PUSH` instruction are passed via the stack.

Figure 6.7 shows the state of the normal data stack and the corresponding shadow stack below the snippet. SP denotes the stack pointer before the instruction is executed. The `SLOAD` instruction will pop an address A from the stack, load the value V (referenced by A) from storage, and then push it onto the stack. Since the `SLOAD` instruction is defined as a taint source, the taint engine will create a new *DependsOnStorage* taint, which we denote as τ_s. This taint is assigned to the value V by pushing it onto the shadow stack. Note that in this case, V was not previously assigned a taint.

The instruction `LT` (*less-than*) compares the value loaded from storage with the value C that was previously pushed on the stack. This comparison decides whether the conditional jump should be taken. Since the `LT` instruction takes two parameters from the stack (V and C), the taint engine also pops two taints from the shadow stack (τ_s and τ'). The result of the comparison is then tainted with both taints (τ_s and τ'), so the taint engine pushes a merged taint (τ_s, τ') to the shadow stack.

The `PUSH2` instruction then pushes a 2-byte constant to the stack, which is assigned an empty taint $\tau_\emptyset$. Finally, the `JUMPI` instruction takes a code pointer (`dst`) and a boolean condition as parameters from the stack. Since `JUMPI` is a taint sink, the taint engine will check the taints associated with the boolean condition. If this value is tainted with the τ_S taint, it will compute the original storage address A based on the taint. At this point, we know that the value at storage address A influenced the control-flow decision. Hence, we add it to the set of control-flow influencing storage addresses, which is passed to the attack detection component later on.

Using the taint engine, Sereum records the set of storage addresses that reference values that influence control-flow decisions. This set of addresses is then forwarded to the attack detection component once the contract finishes executing.

Attack Detection

We lock the write access to storage addresses that influence control-flow decisions to detect reentrancy attacks. During the execution of a contract, the taint engine detects and records storage addresses, which are loaded and then influence the outcome of a control-flow decision. As described in Section 6.3, Sereum uses a dynamic call tree to compute the set of variables that are locked for writing. Sereum builds a so-called dynamic call tree during the execution of a transaction. This tree contains a node for every invocation of a contract during the transaction. The dynamic call-tree records how the call stack of the transaction evolves over time, i.e., it is essentially a combination of all call stacks at any point in time combined into a single data structure. The ordering of the child nodes in the dynamic call tree corresponds to the order of execution during the transaction. The depth of the node in the tree corresponds to the depth in the call stack, i.e., the time when a contract was invoked. Sereum updates the dynamic call tree whenever a contract issues or returns from an

```
1 SLOAD
2 LT
3 PUSH2 $dst
4 JUMPI
```

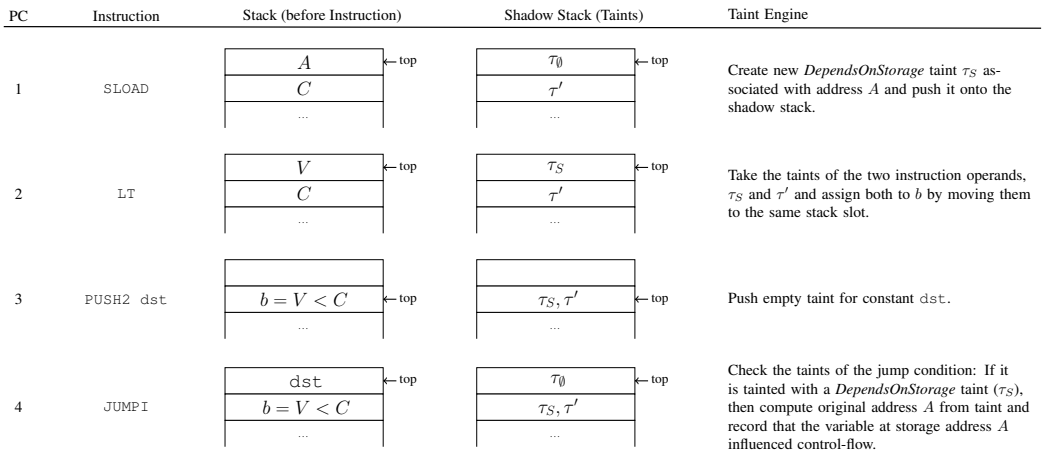


Figure 6.7: Taint propagation example for an EVM snippet that implements a high-level solidity *if-statement* with a conditional branch. The `SLOAD` Instruction in line 1 indirectly influences the control-flow decision in the `JUMPI` instruction in line 4 as it is used as a parameter in the `LT` instruction. `LT` performs a *less-than* comparison between the first and second operands on the stack. The taint engine propagates the taints τ through the executed instructions and stores them on a shadow stack. The condition for the conditional jump b depends on the values C and the value V, which was loaded from storage address A. SP is the current stack pointer, pointing to the top of the data stack.

external call. When the called contract completes execution, the set of control-flow influencing variables is retrieved from the taint engine and stored as associated data in the corresponding node of the call tree.

Sereum locks only the set of variables used during an external call as part of a control-flow decision or as a parameter to another call. We call this type of variable a *critical variable*. To compute this set, Sereum traverses the dynamic call tree starting from the node corresponding to the current execution. During traversal, Sereum searches for nodes, which were part of executions of the same contract. When Sereum finds such a node, it retrieves the set of critical variables previously recorded by the taint engine. Sereum performs this traversal and an update of the set of locked critical variables after every external call. Whenever a contract attempts to write to the storage area, i.e., executes the `SSTORE` instruction, Sereum intercepts the write and first checks whether the address is locked. If the variable is locked, Sereum reports a reentrancy attack and then aborts the execution of the transaction. This results in the EVM unwinding all state changes and Ether transfers.

6.5 Evaluation

In this section, we evaluate the effectiveness and performance of SEREUM based on existing Ethereum contracts deployed on the Ethereum *mainnet*. Since our run-time analysis is transparently enabled for each execution of a contract, we re-execute the transactions that are stored on the Ethereum blockchain. We compare our findings with state-of-the-art academic analysis tools such as Oyente [Luu+16] and Securify [Tsa+18]. The version of Securify, which was available at the time the study was conducted, was only available through a web interface and did not support submitting bytecode contracts anymore. Therefore, we were not able to test all contracts with Securify. Furthermore, we do not compare with Mythril [Conc] and Manticore [Mos+19] as they follow the detection approach of Oyente (symbolic execution). We also conduct experiments based on the three new reentrancy attack patterns we introduced in Section 6.1—effectively demonstrating that only SEREUM is able to detect them all.

6.5.1 Identifying Attacks on Mainnet

We first connect our SEREUM client with the public Ethereum network to retrieve all the existing blocks while keeping as many intermediate states in the cache as possible. Transaction re-execution requires the state of the context block, which is saved as nodes in the state Patricia tree of the Ethereum blockchain. We run the *geth* (Go Ethereum) client with the options sync mode *full*, garbage collection mode *archive*, and assign as much memory as possible for the cache. During the block synchronization process, the taint tracking is disabled to ensure that the client preserves the original state at each block height.

We then replay the execution of each transaction in the blockchain available to us. We performed two evaluation runs of SEREUM. The first initial evaluation re-executed all blocks up to block number 4 500 000 (November, 6th 2017). However, this version of SEREUM did not yet support detection of *unconditional reentrancy*. In a second run, we re-executed all blocks until block number 9 069 000 (December, 8th 2019). We perform a more detailed analysis of the contracts discovered in the first run until block number 4 500 000. Note that we skip those blocks which were targets of denial-of-service attacks as they incur high execution times of transactions [Wil16]. We replay the transactions using the *debug* module of the *geth* RPC API. This ensures that our replay of transactions does not affect the public saved blockchain data. We also retrieve an instruction-level *trace* of the executed instructions and the corresponding storage values during the transaction execution. This allows us to step through the contract's execution at the granularity of instructions.

We enable the taint tracking option in SEREUM during the transaction replay to evaluate whether a transaction triggers a reentrancy attack pattern. If a reentrancy attack is detected, an exception will be thrown, the execution of the transaction gets invalidated, and an error will be reported via the API. SEREUM will then return the instruction trace up to the point where the reentrancy attack is detected.

All in all, we re-executed 597 629 235 transactions, out of which SEREUM flags 312 617 (0.052 %) of them as reentrancy violation. These transactions target 368 different accounts. However, we identify only 250 hash-distinct contracts in this set.

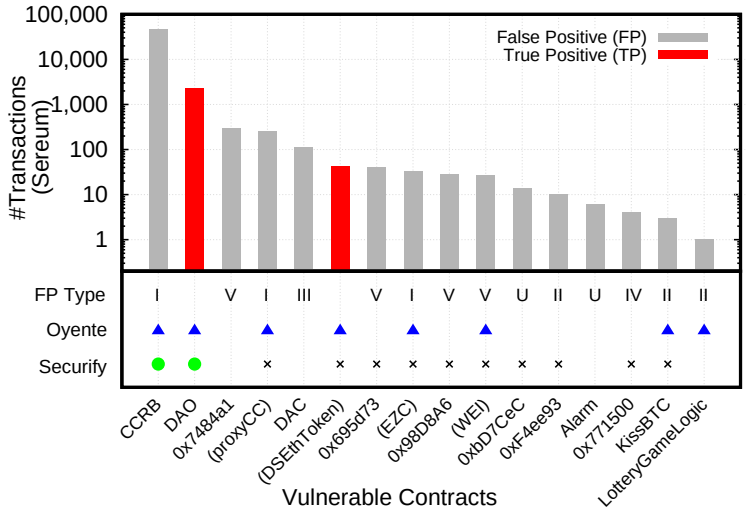


Figure 6.8: The top plot shows the number of detected transactions triggering the reentrancy vulnerability in the flagged contracts. Each contract is categorized by its false positive type described in Section 6.6.1. Type I corresponds to "lack of field-sensitivity", Type II "storage deallocation", Type III "constructor callbacks", Type IV "tight contract coupling", Type V "manual reentrancy locking", and U for Unknown. The contract name is shown for those where source code is available. Contracts in parenthesis are known token contracts at http://etherscan.io although source code is not available. The bottom plot shows how the tools Oyente [Luu+16] and Securify [Tsa+18] handle this subset of contracts. Since the last public version of Securify requires source code, we depict a cross for those (bytecode) contracts we were not able to evaluate.

Furthermore, the majority of transactions are caused by very few contracts. 77.38 % of the transactions are due to 3 different contracts.

In the first part of the evaluation, which covers up to block number 4 500 000, we discovered that many contracts that we flag as vulnerable are created by the same account and share the same contract code but are only instantiated with different parameters. We consider these contracts as being *identical*. More specifically, we found three groups of identical contracts involving 21, 4, and 3 contracts, respectively. Furthermore, we identify many contracts that are essentially duplicates but feature a different code hash. These contracts execute the same sequence of instructions that only differ in the storage addresses. We consider these contracts as *alike* contracts. In total, we found two groups of similar contracts of size 10 and 3, respectively. As a result, Sereum detected 16 identical or alike contracts that are invoked by transactions matching the reentrancy attack pattern. For 6 out of these 16 contracts, the source code is available on http://etherscan.io, thus allowing us to perform a detailed investigation on why they have been flagged. In the remainder of this section, we present our manual analysis of these contracts. We check whether a violating transaction resembles a

reentrancy attack pattern and whether the code of the concerned contract suffers from reentrancy vulnerability that could potentially be exploited.

For contracts with Solidity source code, we perform source code reviews and check the contract logic that is triggered by the transaction input to identify reentrancy attacks manually. We use the transaction trace as a reference to follow the control flow and observe external calls to other contracts or accounts. For contracts with no source code, we cannot fully recover the semantics of the contract for detected inconsistent state updates. In this case, we use the transaction trace and the *ethersplay* [Cry] disassembler tool to partially reverse-engineer the contracts.

Based on our investigation up to block number 4 500 000, we can confirm that two contracts were actually exploited by means of a reentrancy attack. One of them is the known "The DAO" [DAO] attack attributing to 2294 attack transactions. Note that we consider *TheDarkDAO* [dDAO] and "The DAO" [DAO] contract as being identical. The second case involves a quite unknown reentrancy attack. It occurred at contract address `0xd654bDD32FC99471455e86C2E7f7D7b6437e9179`, and we attributed 43 attack transactions to this incident. After reviewing blog posts and GitHub repositories related to this contract [MKRh; MKRr], we discovered that this contract is known as *DSEthToken* and is part of the *maker-otc* project. This series of attack transactions were initiated by the contract developers after they discovered a reentrancy vulnerability. Since the related funds were drained by (benign) developers, the Ethereum community paid less attention to this incident. Based on our manual analysis of the contracts up to block number 4 500 000, we can approximate the rate of false alarms of Sereum. We compute a false alarm rate of 0.06 % across all the re-run transactions up until block number 4 500 000. Figure 6.8 shows the number of transactions that match the reentrancy attack pattern flagged by Sereum. Some of the results reflect false positives, which we discuss in detail in Section 6.6.1.

We also observe that Oyente flagged 8 of these contracts as vulnerable to reentrancy attacks. Some contracts were not detected by Oyente since Oyente does not consider any of the advanced reentrancy attacks we discussed in Section 6.1. During our analysis, we noticed that in some cases, Oyente warned about reentrancy problems, which are only exploitable with a cross-function reentrancy attack. However, we believe this is due to Oyente incorrectly detecting a same-function reentrancy vulnerability. Apart from the six false alarms in our test set, the analysis performed by previous work [GMS18; Tsa+18] demonstrated that reentrancy detection in Oyente is quite imprecise and features a high number of false alarms and also missed bugs.

With respect to Securify, the latest version of Securify requires the source code of a contract, thereby impeding us from evaluating all contracts. Therefore, we have only examined the contracts whose source code is available. Securify defines a very conservative violation pattern for reentrancy detection that forbids any state update after an external call.

Detecting Further Attacks In our second evaluation run up to block number 9 069 000, Sereum identifies several additional reentrancy attacks. For example, Sereum detects the known attack against the *SpankChain LedgerChannel* contract [Spa18]. Furthermore, Sereum correctly identifies several attacks against test contracts that

are intentionally vulnerable to reentrancy but were deployed to the main Ethereum blockchain nevertheless.

Torres et al. [TSS19] studied the phenomenon of *honeypot contracts*. These contracts seem to be vulnerable according to the source code submitted to the *etherscan* service, but their real deployments break exploitation. Such honeypot contracts are typically a form of scam because they require an initial investment of Ether before attempting exploitation. Essentially, the creator of the honeypot scams malicious actors that attempt to exploit simple vulnerabilities. We identify several attempted attacks against honeypot contracts with SEREUM. Although the honeypot contracts thwart exploitation themselves, SEREUM's reentrancy locking still identifies and flags the attempted reentrant transaction.

6.5.2 Detection Capabilities

We evaluated our new reentrancy attack patterns (see Section 6.2). For each contract, we crafted one attack transaction for SEREUM to perform the check: SEREUM successfully detects all attack transactions to the three vulnerable contracts. Table 6.1 shows an overview of various tools tested against the vulnerable contracts for the new reentrancy attack patterns. As discussed earlier, neither Oyente, Securify, nor Manticore can detect delegated and create-based reentrancy vulnerabilities. While Oyente does not detect the cross-function reentrancy attack, Securify is able to detect it due to its conservative policy. Similarly, Mythril detects cross-function and create-based reentrancy because it utilizes a similar policy to Securify, which is extremely conservative and, therefore, also results in a high number of false positives. ECFChecker [Gro+18] detects the cross-function reentrancy attack. However, during our evaluation, we crafted another contract, which is vulnerable to cross-function reentrancy, but was not detected by ECFChecker. Recall that the delegated reentrancy attack cannot be detected by any existing static offline tool as it exploits a dynamic library that is either not available at analysis time or might be updated in the future. However, a dynamic tool, such as ECFChecker or SEREUM, can detect the delegated reentrancy attack. The create-based reentrancy attack is not detected by any of the existing analysis tools, as the instruction `CREATE` is currently not considered as an external call by none of the existing analysis tools.

In general, we argue that SEREUM offers the advantage of detecting actual reentrancy attacks and not just possible vulnerabilities. In contrast to previous work on static analysis [Luu+16; Tsa+18], this makes it feasible for us to determine exactly whether an alarm is a true or false alarm. Moreover, some of the contracts are not flagged by Oyente and Securify as these do not cover the full space of reentrancy attacks. As such, they naturally do not raise false alarms for contracts that violate reentrancy patterns that are closely related to the delegated and create-based reentrancy (i.e., Type III and IV).

Table 6.1: Comparison of reentrancy detection tools subject to our test cases for the advanced re-entrancy attack patterns. Tools marked with ● support detecting this type of re-entrancy, while tools marked with ○ do not support detecting this type of re-entrancy. Tools with an overly restrictive policy are marked with ◐.

Tool	Version	Cross-Function	Delegated	Create-based	Unconditional
Oyente	0.2.7	○	○	○	●
Mythril	0.19.9	◐	○	◐	●
Securify	2018-08-01	◐	○	○	●
Manticore	0.2.2	●	○	○	●
ECFChecker	geth1.8port	●	●	○	●
Sereum	v1	●	●	●	○
Sereum	v2	●	●	●	●

6.5.3 Performance and Memory Overhead

At the time of evaluation, there were no benchmarks, consisting of realistic contracts, available for EVM implementations. As such, we measured the performance overhead by timing the execution of a subset of blocks from the Ethereum blockchain. We sampled blocks from the blockchain, starting from 4 600 000, 4 500 000, 4 400 000, 4 300 000, and 4 200 000, we use 10 consecutive blocks each. We re-execute those blocks in batch while accounting only for the EVM's execution time. We perform one run with plain *geth*, on which Sereum is based, and one with Sereum with attack detection enabled. For the performance evaluation, we do not consider those transactions, which Sereum flags as a reentrancy attack. Sereum aborts those transactions early, which can result in a much shorter execution time compared to the normal execution.

Table 6.2 shows the runtime measurements of *geth* and Sereum in comparison. Sereum incurs an average runtime overhead of 9.6 %. We measured the performance overhead of Sereum, compared with plain *geth* when running 50 blocks in batch. Here, we average the runtime over 10 000 runs of the same 50 blocks. We benchmarked on an 8-core Intel(R) Xeon(R) CPU E5-1630 v4 with 3.70GHz and 32 GB RAM. Sereum incurred a mean overhead of 217.6 ms ($\sigma = 100.9$ ms) or 9.6 %. While measuring the timing of the executed transactions, we additionally measured the memory usage of the whole Ethereum client. We used Linux *cgroups* to capture and measure the memory usage of Sereum and all subprocesses. We sample the memory usage every second while performing the runtime benchmarks. During our benchmark, Sereum required on average 9767 Mbyte of memory with active attack detection, while the plain *geth* required 9252 Mbyte.

This shows that Sereum can effectively detect reentrancy attacks with negligible overhead. In fact, the actual runtime overhead is not noticeable. The average time until the next block is mined in 14.5 seconds and contains 130 transactions on average (between Jan 1, 2018 and Aug 7, 2018). Given our benchmark results, a rough estimate of EVM execution time per block is 0.05 seconds, with Sereum adding 0.005 seconds overhead. Compared to the total block time, the runtime overhead of Sereum is therefore not noticeable during normal usage.

Table 6.2: Performance overhead of Sereum version 1, compared with the plain *go-ethereum* client when running 50 blocks in batch and averaged over 10000 repeated trials.

	Average	σ
go-ethereum Client	2277.0 ms	146.7 ms
Sereum	2494.5 ms	174.8 ms
Overhead	217.6 ms	100.9 ms

6.6 Limitations

In this chapter, we discuss the limitations of Sereum. During our thorough evaluation, we discovered several patterns that caused false alarms. We analyze those in detail and discover several code patterns that tend to cause false alarms. We describe these false alarm patterns in Section 6.6.1. Some of these false alarm patterns, such as storage deallocation, can also be the cause of true reentrancy attacks. As such, some patterns must be accepted as a source of false alarms. We suggest that smart contract authors instead avoid such patterns as they hinder analysis. Other patterns, like the lack of field sensitivity, are something that can be improved upon with better analysis, e.g., taint tracking at byte granularity. Finally, we also discuss a recently discovered reentrancy bug that is missed by Sereum in Section 6.6.2.

6.6.1 Analysis of False Alarms

While investigating the contracts that triggered the reentrancy detection of Sereum, we discovered code patterns in deployed contracts (see Figure 6.8), which are challenging to accurately handle for any off-line or run-time bytecode analysis tool. These patterns are the root cause for the rare false alarms we encountered during our evaluation of Sereum.

However, since these code patterns are not only challenging for Sereum but for other existing analysis tools such as Oyente [Luu+16], Mythril [Conc], Securify [Tsa+18], or any reverse-engineering tools operating at EVM bytecode level [Cry; Zho+18], we believe that a detailed investigation of these cases is highly valuable for future research in this area. Our investigation also *reveals for the first time* why existing tools suffer from false alarms when searching for reentrancy vulnerabilities. In what follows, we reflect on the investigation of the false positives that we encountered.

I. Lack of Field-Sensitivity on the EVM Level Some false positives are caused by a lack of information on fields at the bytecode level for data structures. Solidity supports the keyword `struct` to define a data structure that is composed of multiple types, e.g., Figure 6.9 shows a sample definition of a `struct` S of size 32 bytes. Since the whole type can be stored within one single word in the EVM storage area, accessing either of the fields a or b ends up accessing the same storage address. In other words, on the EVM bytecode level, the taint-tracking engine of Sereum cannot differentiate the access to fields a and b. This leads to a problem called *over-tainting*, where taints spread to unrelated values and, in turn, cause false positives. Notice that this problem affects all analysis tools working on the EVM bytecode level. Some static analysis tools [Gre+19; Sui17] use heuristics to detect the high-level types in Ethereum bytecode. The same approach could be used to infer the types of different fields of a packed data structure. However, for a run-time monitoring solution, heuristic approaches often incur unacceptable runtime overhead without guaranteeing successful identification. To address this type of false positive, one would either require the source code of the contract or additional type information on the bytecode level.

```
struct S {
 int128 a;   // 16 bytes
 int128 b;   // 16 bytes
}            // total: 32 bytes
// struct access in solidity, e.g.
S.a = 0x42;
// compiles down to code roughly equivalent to the following evm assembly code.
assembly {
 // load b
 b := and(sload(S), ((2 << 127) - 1))
 // write new value for a and old value of b
 sstore(S, or((0x42 << 128), b))
}
```

Figure 6.9: Solidity *struct*, where both a and b are at the same storage address. Therefore, any update to a or b includes loading and writing also the other.

```
mapping (uint => uint) M; // a hash map
// delete entry from mapping
delete M[id];
// on the EVM level this is equivalent to
M[id] = 0;
```

Figure 6.10: Solidity storage delete is equivalent to storing zero.

II. Storage Deallocation The EVM *storage* area is basically a key-value store that maps 256-bit words to 256-bit words. The EVM architecture guarantees that the whole *storage* area is initialized with all-zero values and is always available upon request. More specifically, no explicit memory allocation is required, while memory deallocation simply resets the value to zero. This poses a problem at the bytecode level: a memory deallocation is no different from a state update to value 0, though the semantics differ, especially when applying the reentrancy detection logic. Consider the example of a map M in Figure 6.10. When the contract deallocates the element indexed by id from M (*delete* from a map), it basically has the same effect as setting the value of $M[id]$ to 0 at the bytecode level. Here, the Solidity compiler will emit nearly identical bytecode for both cases. We encountered a contract[2] presenting this case which leads to a false alarm.

However, this pattern does not necessarily always lead to false alarms. In fact, the very same pattern can also lead to real reentrancy attacks. For example, the *SpankChain LedgerChannel* [Spa18] was vulnerable to a reentrancy bug that was because of this pattern. Figure 6.11 shows the vulnerable function. As such, we conclude that while this is a common pattern for false alarms, it is important to cover this case for accurate reentrancy detection. In fact, any false alarm due to this pattern is a sign of bad coding practices and also potential reentrancy attacks.

[2]Contract address: 0x6777c314b412f0196aca852632969f63e7971340

```
function LCOpenTimeout(bytes32 _lcID) public {
  /* ... */
  if (Channels[_lcID].initialDeposit[0] != 0) {
    // ETHER TRANSFER
      Channels[_lcID].partyAddresses[0].transfer(
          Channels[_lcID].ethBalances[0]
      );
  }
  if (Channels[_lcID].initialDeposit[1] != 0) {
      // EXTERNAL CALL
      require(
          Channels[_lcID].token.transfer(
              Channels[_lcID].partyAddresses[0],
              Channels[_lcID].erc20Balances[0]
          ),
          "CreateChannel: token transfer failure"
      );
  }

  /* ... */

  // STATE UPDATE
  delete Channels[_lcID];
}
```

Figure 6.11: Vulnerable function of the SpankChain LedgerChannel contract that was exploited based on the fact that the *delete* is performed as the last action. The function repeatedly transfers Ether when reentered because the *channel* structure is not deleted before the external call.

III. Constructor Callbacks SEREUM considers calls to the constructor of contracts to be the same as calls to any other external contract. This allows SEREUM to detect create-based reentrancy attacks. However, detecting create-based reentrancy comes at the cost of some false positives. During our evaluation[3], we noticed that sub-contracts created by other contracts, tend to call back into their parent contracts. Usually, this is used to retrieve additional information from the parent contract: the parent creates the sub-contract, the sub-contract re-enters the parent contract to retrieve the value of a storage variable, and that same variable is then updated later by the parent. Consider the example in Figure 6.12, where contract A creates a sub-contract B. While the constructor executes, B re-enters the parent contract A, which performs a control-flow decision on the *funds* variable. This results in SEREUM locking the variable *funds*. Since no call to another potentially malicious external contract is involved, this example is not exploitable via reentrancy. However, SEREUM detects that the *funds* variable is possibly inconsistent due to the deferred state update. A malicious contract B could have re-entered A and modified the *funds* variable in the meantime.

[3]Contract address 0xFBe1C2a693746Ccfa2755bD408986da5281c689F

```
contract A {
  mapping (address => uint) funds;
  // ...
  function hasFunds(address a) public returns(bool) {
    // funds is used for control-flow decision
    if (funds[a] >= 1) { return true; }
    else { return false; }
  }
  function createB() {
    B b = new B(this, msg.sender);
    // ...
    // update state (locked due to call to hasFunds)
    funds[msg.sender] -= 1;
  }
}
contract B {
  constructor(A parent, address x) {
    // call back into parent
    if (parent.hasfunds(x)) { /* ... */ }
  }
}
```

Figure 6.12: Constructor callback: The sub-contract B calls back (re-enters) into the *hasFunds* function of the parent contract A. This type of false positive is similar to the create-based reentrancy attack pattern.

We argue that this constructor callback pattern should be avoided by contract developers. All necessary information should be passed to the sub-contract's constructor such that no reentrancy into the parent contract is needed. This not only avoids false positives in SEREUM but also decreases the gas costs. External calls are one of the most expensive instructions in terms of gas requirements, which must be paid for in Ether and, as such, should be avoided as much as possible.

IV. Tight Contract Coupling During our evaluation, we noticed a few cases where multiple contracts are tightly coupled, resulting in overly complex transactions, i.e., transactions that cause the contracts to be re-entered multiple times into various functions. This suggests that these contracts have a strong interdependency. Since SEREUM introduces locks for variables that can be potentially exploited for reentrancy and is not aware of the underlying trust relations among contracts, it reports a false alarm when a locked variable is updated. We consider these cases as an example of bad contract development practice since performing external calls is relatively expensive in terms of gas, and as such also Ether, and could be easily avoided in these contracts. That is, if trusted contracts have an internal state that depends on the state of other trusted contracts, we suggest developers keep the whole state in one contract and use safe library calls instead.

```
1  mapping (address => uint) private balances;
2  mapping (address => bool) private disableWithdraw;
3  // ...
4  function withdraw() public {
5  ① if (disableWithdraw[msg.sender] == true) {
6        // abort immediately and return error to caller
7        revert();
8     }
9     uint amountToWithdraw = balances[msg.sender];
10
11 ② disableWithdraw[msg.sender] = true;
12 ③ msg.sender.call.value(amountToWithdraw)();
13 ④ disableWithdraw[msg.sender] = false;
14    // state update after call
15    userBalances[msg.sender] = 0;
16 }
17 // ...
```

Figure 6.13: Manual locking to guard against reentrancy.

V. Manual reentrancy Locking To allow expected and safe reentrancy, a smart contract can manually introduce lock variables (i.e., a mutex) to guard the entry of the function. In Figure 6.13), *disableWithdraw* enables a lock at ② before making an external call at ③ . The lock is reset after the call at ④ . This prevents any potential re-entrance at ① . Hence, even though the *balance* is updated after the external call, the contract is still safe from reentrancy attacks.

However, the access pattern to these lock variables during reentrant calls matches an attack pattern, i.e., the internal state (the lock variable) that affects the control flow in subsequent (reentered) invocation of the contract, is updated subsequently (at ④). Operating at the bytecode level, it is challenging to distinguish the benign state updates of *locks* from those of critical variables such as *balances*. Note that manual locking is an error-prone approach as it could allow an attacker to reenter other functions of the same contract unless the entry of every function is guarded by the lock. In contrast, SEREUM automatically introduces locks for all possibly dangerous variables (detected via taint tracking) across all functions, thereby removing the burden from developers to manually determine all possible vulnerable functions and critical variables.

VI. Bounds-Checking of Dynamic Arrays One of the largest sources of false alarms we discovered during our extended evaluation was the bounds-checking code of variably-sized arrays. Most notably, the popular *BlockChainCuties* contract was responsible for many false alarms. These bounds-checks are generated by the Solidity compiler and are not directly visible to the developer. Figure 6.14 shows the relevant code-snippets from the *BlockChainCuties* contract. The contract is really split into two contracts, similar to the tight contract coupling issue (issue IV.), and there are frequent callbacks between the two contracts. Most importantly, the second contract will perform a call that triggers a bounds check before accessing an array. On the level of the Solidity

```
// reentrant call to this functions sets a write-lock
function getGeneration(uint40 _id) public view
    returns (uint16 generation)
{
  // solidity generates a bounds-check here, which is
  // implemented as an EVM-level jump.
  Cutie storage cutie = cuties[_id];
  generation = cutie.generation;
}
// [...]
// ...which is violated when the array is modified later
uint256 newCutieId256 = cuties.push(_cutie) - 1;
// [...]
```

Figure 6.14: Reentrant callback to the *getGeneration* function in the *BlockChainCuties* contract, will set a write-lock that is later violated.

source code, there is no conditional branch visible. However, on the EVM level, Solidity generates a conditional branch for the bounds-check. The generated code loads the length of the array from the storage area, compares it to the index passed by the user, and if it is out of bounds, the conditional branch will jump to the error reporting code. On the EVM level, there is no difference between the length and a storage value with the semantics of a user balance. As such, SEREUM sets a write-lock on the length of the dynamic array. However, if the array is later modified, SEREUM will report a potential reentrancy attack. This pattern is particularly tricky to assess because there is no clear root cause for the write-lock on the source code level.

6.6.2 Missed Reentrancy Patterns

While most reentrancy attacks attempt to bypass a security check, i.e., something represented as a conditional branch, we already discussed an outlier to this rule: unconditional reentrancy. Sometimes, even critical functions are so small that they can be implemented without any conditional branch at all. Similar to this pattern, Bose et al. [Bos+22] identified a reentrancy pattern that is quite similar to traditional *double-fetch* vulnerabilities [Wan+17]. Figure 6.15 show an example for such a vulnerability [Bos+22], which we accordingly call *double-fetch reentrancy*. Here, there are two consecutive uses, i.e., read accesses, of a storage variable, and between both uses, there is an external call. The implicit assumption is that the value obtained by the storage reads is the same. During normal execution, this is a reasonable assumption if there is no write to that storage address. However, since there is an external call between both reads, this assumption can be violated.

In the example in Figure 6.15, the code of the victim fetches the variable twice, assuming it has the same value. However, during the external call, the attacker maliciously reenters the victim contract in the *updateSplit* function. Now the value of the split changed, and the attacker is able to manipulate the following computation of the splits to gain Ether.

```
1  // [Step 1]: Set split of 'a' (id = 0) to 100(%)
2  // [Step 4]: Set split of 'a' (id = 0) to 0(%)
3  function updateSplit(uint id, uint split) public{
4     require(split <= 100);
5     splits[id] = split;
6  }
7
8  function splitFunds(uint id) public {
9     address payable a = payee1[id];
10    address payable b = payee2[id];
11    uint depo = deposits[id];
12    deposits[id] = 0;
13
14    // [Step 2]: Transfer 100% fund to 'a'
15    // [Step 3]: Reenter at updateSplit
16    // first load from storage
17    a.call.value(depo * splits[id] / 100)("");
18
19    // [Step 5]: Transfer 100% fund to 'b'
20    // second load from storage: DOUBLE-FETCH
21    b.transfer(depo * (100 - splits[id]) / 100);
22 }
```

Figure 6.15: Example for a double-fetch reentrancy vulnerability [Bos+22]

This attack remains undetected by Sereum. The problem is that Sereum utilizes a different model of reentrancy: one that focuses on write access. Sereum assumes that the reentrant call will set a write-lock due to a storage read, and the non-reentrant call will violate the write-lock with a storage write operation. However, in this case, the storage write operation is performed during a reentrant call, while the storage read operation is in the non-reentrant call.

6.7 Related Work

Reentrancy has emerged as one of the major challenges for smart contract security. Years after "The DAO" incident, we still observe major incidents and losses due to reentrancy bugs [REVST; CREAM; Tor+21b]. Naturally, many different analysis methods to identify reentrancy issues were proposed. The types of methods to tackle the reentrancy problem range from formal verification, static analysis, bounded model checking, symbolic execution, and fuzzing.

Zeus [Kal+18] translates Solidity contract into the *LLVM* intermediate language and then utilizes a stock verification framework, in this case *Seahorn*. The verification framework then identifies potential violations of the assertions inserted based on a custom security policy. Zeus also identifies reentrancy attacks, but it only scales to the *same-function* reentrancy pattern. The problem is that it does not explicitly model reentrancy during the verification process but works around it by inserting spurious calls into the analyzed *LLVM* bitcode for analysis purposes. This approach does not scale to other reentrancy patterns due to combinatorial explosion. However, especially

newer reentrancy attacks are often due to cross-function attacks [REVST; CREAM; Tor+21b]. In contrast, Schneidewind et al. [Sch+20b] provides a provably sound static analysis framework tailored towards the EVM. This static analysis framework allows to verify the *single entrancy* property for EVM smart contracts. Similarly, *KEVM* [Hil+17] defines executable formal semantics for EVM bytecode in the $\mathbb{K}$-framework and presents an accompanying formal verification tool.

Securify [Tsa+18] is a static analysis tool that infers semantic facts about smart contracts using a Datalog solver [Sch+16]. The Datalog solver then attempts to show whether a predefined compliance pattern or violation pattern is satisfied, thereby proving the absence or presence of certain vulnerabilities. To detect reentrancy, Securify uses the *no-write after-call* analysis. Essentially, this type of analysis enforces a variant of the *checks-effects-interactions* pattern. Securify will flag any state write after an external call as potential reentrancy. Naturally, this will lead to many false alarms in complex contracts, where this pattern is often impossible to avoid. For example, it would not be practical to model *create-based reentrancy* with this security policy. Typically, the parent contract must save the address of the newly created child contract. Naturally, this is only possible after issuing the *create* instruction, and also after the potential reentrant call, as the *create* instruction itself will return the address. Even though the *no write after call* security policy exhibits a high number of false alarms, this type of analysis is used in various analysis tools [FGG19; Conc].

Two hybrid fuzzing tools utilize dynamic analyses similar to Sereum's policy to determine potential reentrancy. Confuzzius identifies a sequence of instructions in an execution trace: (1) a `SLOAD`, (2) an external call, and (3) a `SSTORE` to the same address as the prior `SLOAD`. This detection method is more similar to Sereum than forbidding all writes after calls. Similar to Sereum, Confuzzius attempts to detect writes that could cause inconsistent state during reentrant executions. However, Confuzzius does not track data dependencies as does Sereum. While this allows Confuzzius to naturally cover unconditional reentrancy, it does so at the cost of a higher number of false alarms. Very similar to Sereum, *Smartian* leverages a taint analysis during hybrid fuzzing and concolic execution to identify reentrancy bugs [Cho+21]. Smartian tracks state variables using taint analysis that affect ether transfers. If the state variable is updated after the call, Smartian reports a reentrancy bug. However, Smartian's reentrancy detection must be seen as an extension to the one used by Confuzzius, as it does not require an actual reentrant execution to trigger. In contrast, Sereum will only report reentrancy if it actually observes a reentrant execution.

ECFChecker [Gro+18] is a dynamic analysis tool that does not detect reentrancy vulnerabilities, but defines a new attribute, *Effectively Callback Free (ECF)*. An execution is ECF when there exists an equivalent execution without callbacks that can achieve the same state transition. If all possible executions of a contract satisfy ECF, the whole contract is considered as featuring ECF. Non-ECF contracts are thus considered as vulnerable to reentrancy, as callbacks can affect the state transition upon contract execution. Proving the ECF property with static analysis was shown to be undecidable in general. However, Grossman et al. [Gro+18] also developed a dynamic checker that can show whether a transaction violates the ECF property of a contract. *ECFChecker* has been developed concurrently to Sereum and was, to the best of our knowledge, the only other runtime monitoring tool at that time. Recently,

the ECF property was adapted by the same authors to a constructive version that can be statically verified [Alb+20]. However, the ECF approach does not cover the full space of reentrancy attacks and misses several reentrancy patterns. Furthermore, it is not clear whether the ECF property also covers compositional reentrancy attacks that chain across multiple contracts [Cec+21].

Bose et al. [Bos+22] introduce the symbolic execution tool *Sailfish* that is tailored specifically to uncovering *state inconsistency* bugs. The Sailfish analyzer summarizes both reentrancy and transaction-order dependence under state inconsistency. Sailfish combines static data-dependency analysis, value-summary analysis, and symbolic execution to identify state inconsistency.

6.8 Conclusion

Reentrancy attacks exploit the inconsistent internal state of smart contracts during unsafe reentrant executions, allowing an attacker, in the worst case, to drain all available assets from a smart contract. Previously it was thought that advanced offline analysis tools could accurately detect these vulnerabilities. However, as we show, these tools can only detect basic reentrancy attacks and fail to accurately detect new reentrancy attack patterns, such as cross-function, delegated, and create-based reentrancy. Furthermore, it remains an open research direction on how to protect existing contracts as smart contract code is supposed to be immutable and contract creators are anonymous, which impedes responsible disclosure and deployment of patched contract. We discuss a direction forward in this area in Chapter 7.

To address the particular ecosystem of smart contracts, we introduce a novel runtime smart contract security solution, called Sereum, which exploits dynamic taint tracking to monitor data-flows during smart contract execution to automatically detect and prevent inconsistent state and thereby effectively prevent basic and advanced reentrancy attacks without requiring any semantic knowledge of the contract. By running Sereum on more than 80 million Ethereum transactions involving more than 90 000 contracts, we show that Sereum can identify and prevent reentrancy attacks in deployed contracts with negligible overhead.

Limitation and Future Work Sereum is designed to run in enforcement mode, protecting existing contracts, when Sereum is integrated into the blockchain ecosystem. However, Sereum can also be utilized by smart contract developers in order to identify attacks against their contracts and patch them accordingly. Namely, Sereum can also be executed locally by contract developers that are interested in ensuring the security of their deployed contracts. In fact, this mode is most useful today, as it is unlikely that a tool like Sereum will be integrated into the default Ethereum execution environment. The problem here is that Sereum changes the semantics of the execution environment in rather surprising ways, as we identify in our analysis of false alarms in Section 6.6.1. The fuzzer EF↯CF, which we describe in Chapter 5, is the first fuzzing tool that generates and concretely executes reentrant transactions. However, EF↯CF does not feature explicit detection of reentrancy issues. Instead, it relies on other bug oracles, such as Ether gains, to identify reentrancy vulnerabilities. While this

leads to very accurate results, it does not uncover code quality issues that currently cannot be exploited but should be removed to follow the defense-in-depth principle. To detect reentrancy, Sereum must observe concrete reentrant transactions, something provided by EF↯CF. As such, it would be an interesting future research direction to pair the taint tracking featured in Sereum with the transaction generator of EF↯CF to also detect potential reentrancy bugs improving the code quality of smart contracts. The taint tracking code in Sereum does not add significant overhead with respect to the baseline interpreter. However, for usage in EF↯CF, the taint-tracking engine must be evaluated with respect to the overhead to the high throughput execution approach featured in EF↯CF. As such, it remains an open question on how to integrate taint-tracking-based detection into a high-performance Ethereum fuzzer.

Recently it was found that Sereum does not identify all types of reentrancy. Bose et al. [Bos+22] identify an example of a reentrancy bug that is not picked up by Sereum, a pattern we call *double-fetch reentrancy*. Sereum's model for automatically performing locking does not match this reentrancy pattern. While the taint-tracking approach of Sereum is potent enough to identify various reentrancy issues at runtime, it is not clear how to extend Sereum to capture also this type of attack. Sereum's locking model could be extended to issue read-locks on reentrant write accesses, but it is unclear that this will not impact regular executions even further. As with any mitigation, the number of false alarms of such an extension to Sereum would need to be investigated to assess the overall usefulness of the defense.

We believe that automatic source-level hardening of smart contracts is a viable alternative approach to securing smart contracts against reentrancy attacks [Gie+22]. Source-level hardening is fully compatible with the existing Ethereum infrastructure and tooling and can be deployed to the main Ethereum network. In contrast to tools that operate on the bytecode level, source-level hardening can perform analysis on the source code to obtain semantic information. Furthermore, a source-level hardening approach can be inspected by a developer to remove the overhead of unnecessary checks. As such, hardening at the source-code level could be a more practical solution in the short term.

CHAPTER 7

AUTOMATIC PATCHING OF SMART CONTRACTS

A number of ongoing attacks have fueled interest in the community to enhance the security of smart contracts. In this respect, a number of solutions ranging from devising better development environments to using safer programming languages, formal verification, symbolic execution, and dynamic runtime analysis have been proposed in the last few years [Gro+18; Kal+18; Luu+16; Rod+19]. However, most analysis tools focus on detecting vulnerabilities or proving the absence of a certain type of bug [Gri+20; Kal+18; Luu+16; Mos+19; Tsa+18]. This includes the fuzzing technique we describe in Chapter 5. Dynamic analysis techniques can be integrated directly into the blockchain to mitigate attacks as they happen [Gro+18; Rod+19], which we discuss in Chapter 6. However, it is unclear how to deal with false alarms in these dynamic analyses when integrated directly into the blockchain platform. Furthermore, a post hoc integration into current production blockchain systems is unlikely due to the need for backward compatibility. As such, they cannot be used to protect already deployed legacy contracts on, e.g., the main Ethereum blockchain as it is used today. Since none of the prior method of securing smart contracts are sufficient, a method to upgrade smart contracts that are being deployed to current blockchain systems is needed.

However, the patching lifecycle of smart contracts on the Ethereum blockchain is quite complex. By design, Ethereum smart contracts are immutable, which requires more involved upgrading strategies to work around this. The naive approach, called *migration*, requires the contract owner to deprecate the vulnerable contract, move all funds out of the contract, deploy a new contract, and finally move the funds to the new contract. This is a cumbersome and manual process, which is further exacerbated when the address of the vulnerable contract is referenced by other contracts. Besides the migration pattern, *proxy* contracts have emerged as the de-facto standard for upgradable contracts. The idea is that a truly immutable proxy contract uses an interchangeable and trusted *logic* contract to implement the actual business logic.

With EVMPatch, we address the problem of automated and timely patching of smart contracts to aid developers in instantly taking action on reported smart contract errors. We describe our patching framework EVMPatch that features

a *bytecode-rewriter* for Ethereum smart contracts. Employing bytecode rewriting makes our framework independent of the source programming language and allows our framework to work on unmodified contract bytecode. EVMPatch utilizes the bytecode-rewriting engine to ensure that patches are minimally intrusive and that the newly patched contract is compatible with the original contract. In particular, our framework automatically replays transactions on the patched contract to

1. test the functional correctness of the patched contract with respect to previous transactions pertaining to the contract,
2. identify potential attacks, i.e., developers can determine whether their vulnerable contract has been attacked in the past.

EVMPatch uses a best-effort approach to ensure the introduced patch does not break functionality by testing with previously issued transactions to the contract and optionally also developer-provided unit tests. While such a differential testing approach cannot provide a formal proof of the correctness of the patched contract, it works without requiring a formal specification. Our experiments (see Section 7.4) show that this approach is sufficient in practice to identify broken patches.

Contributions We present the design of EVMPatch, an automated approach to smart contract patching. We show how to implement such a highly automated patching workflow using bytecode-level patches. By applying patches on the bytecode level, EVMPatch is independent of the used programming language/compiler and compiler version. That is, EVMPatch supports any off-the-shelf Ethereum smart contract code. We employ bytecode writing to ensure minimally intrusive patches that are compatible by design with the contract's storage layout. However, as for any approach working on either the binary or bytecode level, we had to tackle several technical challenges (Section 7.3). EVMPatch not only patches a smart contract but also automatically converts the original contract to use the delegatecall-proxy pattern. As such, EVMPatch can automatically deploy newly patched contracts in a fully automated way without requiring any developer intervention using this proxy pattern.

In principle, EVMPatch can support patching of different classes of vulnerabilities (see Section 7.3.4). However, our proof-of-concept implementation targets two major bug classes: access control and integer overflow bugs. The latter has been repeatedly exploited in high-value ERC-20 contracts [Pecc], whereas the former has been abused in the Parity wallet attack [Tec17b]. In our EVMPatch prototype, we fully automate patching integer overflow bugs. To keep the overhead minimal and ensure contract functionality, we selectively harden exactly those integer arithmetic instructions, which are considered to be vulnerable by the existing symbolic execution tool *Osiris* [FSS18]. However, our automated patch generation tool does not depend on one specific analysis tool. In fact, any static or dynamic analysis tool can be easily integrated. With access control issues as an example, we discuss what it takes to integrate the capability to patch a new type of vulnerability into the EVMPatch framework.

To evaluate EVMPatch in terms of performance, effectiveness, and functional correctness, we apply EVMPatch to several real-world vulnerable contracts. To this end, we used the patch testing component of the EVMPatch framework to re-play all existing transactions to the original contract on the patched contract. This

allows us to provide an in-depth investigation of several actively exploited smart contracts, e.g., token burning and history of attack transactions (before and after public disclosure). For a number of contracts we investigated in our evaluation, we found that EVMPatch would have blocked several attacks that happened after public disclosure of the vulnerability. This shows that even though those contracts were officially deprecated, they were still used by legitimate users and exploited by malicious actors. As such, there is an immediate need for tooling, as provided by EVMPatch, which allows the developers of smart contracts to efficiently patch their contracts. Our evaluation also covers important practical aspects such as gas and performance overhead (i.e., the costs for executing transactions in Ethereum). The gas overhead for all our patched contracts was below 0.01 US$ per transaction. Overall, the performance overhead was negligible.

The basis for this chapter is the following publication:
"EVMPatch: Timely and Automated Patching of Ethereum Smart Contracts". ***30th USENIX Security Symposium*, 2021**. Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi

7.1 Background on Patching Smart Contracts

In this section we review the necessary background on upgrading smart contracts in Section 7.1.1 and the challenges and pitfalls of rewriting EVM bytecode in Section 7.1.2.

7.1.1 Upgrading Ethereum Smart Contracts

Ethereum treats the code of smart contracts as immutable once they are deployed on the blockchain, with the exception of the selfdestruct mechanism that destroys a smart contract. To remedy this, the community came up with strategies for upgrading smart contracts [Cona; Nad; Tra18]. Currently, the two most common patterns are the migration pattern and variations of the proxy pattern.

Superficially, the migration upgrade pattern is the most naive pattern and can be applied to many smart contracts. The patched contract is deployed on the blockchain at a separate new address with a fresh and clean state. The administrators of the contract then need to inspect the state of the old contract and manually migrate all stored values to the new contract. Because of this, state migration is specific to the contract and must be manually implemented by the developers of the contract. It requires the contract developers to have access to all the internal state of the old contract and a procedure in the new contract to accept state transfers. To avoid state migration, developers can also use a separate contract as a data storage contract, which is sometimes referred to as the eternal storage pattern [EIP; Nad]. However, this adds additional gas overhead since every time the logic contract needs to access data, the contract must also perform a costly external call into the data storage contract.

A more common strategy is to write contracts with the proxy-pattern, with the most favorable version being the *delegatecall-proxy* pattern. Here, one smart contract is split into two different contracts, one for the code and one for data storage: i) an immutable *proxy contract*, which holds all funds and all internal state but does not implement any business logic; ii) a *logic contract*, which is completely stateless and

implements all the actual business logic, i.e., this contract contains the actual code that governs the actions of the contract. The proxy contract is the entry point of all user transactions. It has immutable code, and its address remains constant over the lifetime of the contract. The logic contract implements the rules that govern the behavior of the smart contract. The proxy contract forwards all function calls to the registered logic contract using the `DELEGATECALL` instruction. This instruction gives the logic contract access to all internal state and funds stored in the proxy contract. To upgrade the contract, a new logic contract is deployed, and its address is updated in the proxy contract. The proxy contract then forwards all future transactions to the patched logic contract. As a result, deploying upgraded contracts does not require any data migration, as all data is stored in the immutable proxy contract. Moreover, the upgrading process is also *transparent* to users, as the contract address remains the same. Although existing blockchain platforms do not provide mechanisms to upgrade smart contracts, the usage of this proxy pattern allows EVMPatch to quickly upgrade a contract with negligible costs (in terms of gas consumption).

7.1.2 Challenges of EVM Bytecode Rewriting

There are several unique challenges that must be solved when rewriting EVM bytecode: we need to handle static analysis of the original EVM bytecode and tackle several particularities of Solidity contracts and the EVM.

Similar to traditional computer architectures, EVM bytecode uses addresses to reference code and data constants in the code address space. For instance, when a contract calls a function, it first pushes the address of that function onto the stack and then exploits the EVM jump instruction to transfer control to that function. Hence, when modifying the bytecode, the rewriter must ensure that address-based references are correctly adjusted. In that sense, rewriting EVM bytecode is similar to rewriting binary code for normal CPU ISAs, such as x86 or ARM. To do so, a rewriter typically employs two static analysis techniques: CFG recovery and data-flow analysis. The latter is necessary to determine which instructions are the sources of any address constants utilized in the code. For the EVM bytecode, two classes of instructions are relevant in this context: code jumps and constant data references.

Code Jumps The EVM features two branch instructions: `JUMP` and `JUMPI`. Both take the destination address from the stack. Handling jumps in the EVM is more challenging compared to traditional architectures, as the target address is not explicitly encoded into the jump instruction. Note that function calls inside the same contract also leverage the `JUMP` instruction. That said, currently, there is *no explicit difference* between local jumps inside a function and calls to other functions. The EVM also features dedicated call instructions, but these are only used to transfer control to a completely separate contract. Hence, they do not require modification when rewriting the bytecode.

Constant Data References The so-called `CODECOPY` instruction is leveraged to copy data from the code address space into the memory address space. A common example use-case is the embedding of large data constants such as strings. Similar to the jump

instructions, the address from which memory is loaded is passed to the `CODECOPY` instruction via the stack.

Handling both types of instructions is challenging due to the stack-based architecture of the EVM. For instance, the target addresses of jump instructions are always provided on the stack. That is, every branch is indirect, i.e., the target address cannot be simply looked up by inspecting the jump instruction. Instead, to resolve these indirect jumps, one needs to deploy data-flow analysis techniques to determine where and which target address is pushed on the stack. For the majority of these jumps, one can analyze the surrounding basic block to trace back where the jump target is pushed on the stack. For example, when observing the instructions `PUSH2 0xdb1; JUMP`, we can recover the jump target by retrieving the address (`0xdb1`) from the push instruction.

However, many contracts contain more complicated code patterns, primarily because the Solidity compiler also supports calling functions internally without utilizing a call instruction. Recall that the EVM call instructions are more akin to remote-procedure calls. For instance, when a contract would utilize a call instruction to call itself, the called function cannot access the stack or the memory address space of the caller, as every call in the EVM is considered as a new transaction and as such starts with a clean stack and memory address space. Furthermore, the call instructions are also rather expensive in terms of gas cost, and as such, compilers such as Solidity minimize their use. The Solidity compiler introduced a concept where functions are marked as *internal* to optimize code size and facilitate code reuse. These functions cannot be called by other contracts (private to the contract) and follow a different calling convention. Since there are no dedicated return and call instructions for internal functions, Solidity utilizes the jump instruction to emulate both. As such, a function return and a normal jump cannot be easily distinguished. This makes it challenging to (1) identify internal functions and (2) build an accurate CFG of the contract.

Consider Figure 7.1, which depicts an excerpt of a typical CFG when Solidity generates code utilizing internal function calls. There are two callers of the internal function, named A and B. Each caller pushes a constant return address onto the stack: A pushes the return address X; B the constant Y. These constants are addresses of basic blocks, where execution resumes once the internal function completes. In this example, both callers push the address of the first basic block of the internal function, dubbed F, onto the stack and utilize the jump instruction to call the internal function.

To emulate the return, an indirect jump instruction is leveraged at the end of the internal function, where the target address is taken from the stack. Depending on the calling context, the final jump of the internal function will either jump back into A or B. Note that, for this example, the surrounding basic block does not contain any corresponding push instruction of the jump target. Instead, the respective push instruction issued at the call site has loaded the return address on the stack. Hence, data-flow analysis is needed to determine push instructions that are leveraged for function returns.

When rewriting an EVM smart contract, both the jump instructions and the code-copy instruction need to be considered in the bytecode rewriter. The obvious strategy to rewrite smart contracts is to fix-up all constant addresses in the code to reflect the new addresses after inserting new instructions or removing old instructions. However, this strategy is challenging because it requires accurate CFG recovery and data-flow analysis,

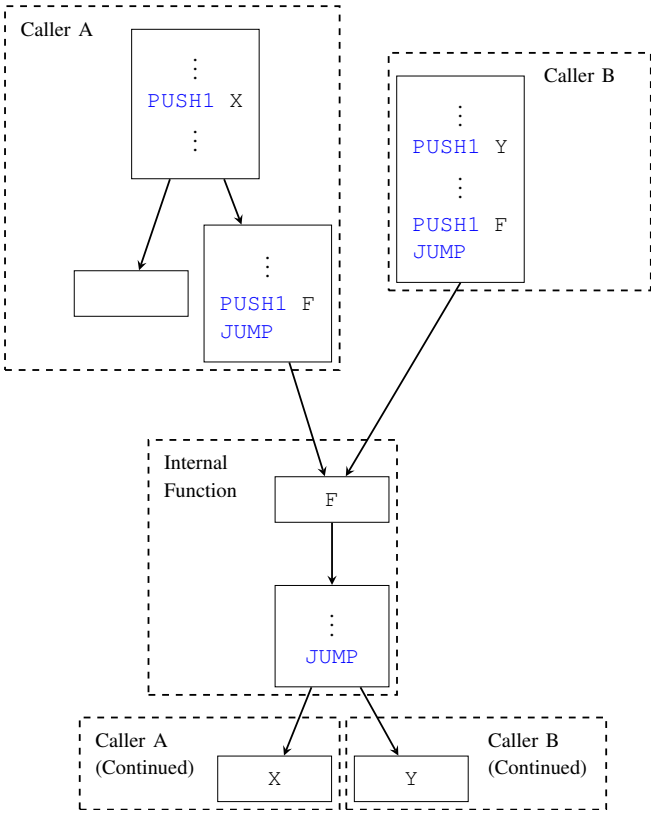


Figure 7.1: A typical control-flow graph, when Solidity utilizes internal function calls.

which needs to deal with particularities of EVM code, such as internal function calls. In the research area of binary rewriting of traditional architectures, a more pragmatic approach has been developed: the so-called trampoline concept [Dav+12; Lau+10]. We utilize this approach in our rewriter and avoid adjusting addresses. Whenever our rewriter must perform changes to a basic block, e.g., inserting instructions, our rewriter replaces the basic block with a trampoline that immediately jumps to the patched copy. Hence, any jump target in the original code stays the same, and all data constants are kept at their original addresses. This approach allows our rewriter to avoid building a complete control-flow graph altogether. We describe this process in more detail in the subsequent section.

7.2 Design of EVMPatch

In this section, we introduce the design of our automated patching EVMPatch framework, which enables developers to patch and harden smart contracts in an automated and timely manner. Our framework operates on unmodified smart contracts and is independent of the source programming language, as it does not require source code. At its core, our framework utilizes a bytecode rewriter to apply minimally intrusive patches to EVM smart contracts. Combined with a proxy-based upgradable

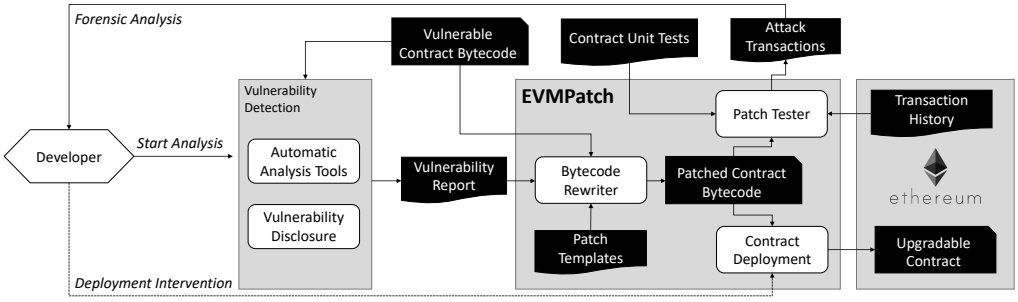


Figure 7.2: Architecture of EVMPatch

smart contract, this bytecode rewriting approach allows the developer to automatically introduce patches and deploy them on the blockchain. One major advantage of this approach is that when new attack types are discovered or bug finding tools improve, the contract can be automatically re-checked, patched, and re-deployed in a short amount of time and with minimal developer intervention.

We envision a setting where EVMPatch is executed on a trusted machine operated by the smart contract developer and is continuously running new and updated vulnerability detection tools. This can also include dynamic analysis tools, which analyze transactions that are not yet included in a block, but are already available to the Ethereum network. Whenever one of the analysis tools discovers a new vulnerability, EVMPatch automatically patches the contract, tests the patched contract, and deploys it.

7.2.1 Design Choices

The proxy pattern makes it possible to easily deploy a patched smart contract in Ethereum. However, it neither generates a patched version nor features functional tests of the patched contract. EVMPatch fills this gap by providing a comprehensive framework that automates many steps in the patching process, such as testing the effectiveness of the generated patch.

For generating patches, there is one fundamental design decision: applying the patch on the source code level or the EVM bytecode level. At first glance, *source-code patching* seems to be the option of choice as developers have access to source code, are able to inspect the source code changes, and can even make adjustments if the automated approach introduces undesired changes. However, when utilizing upgradable contracts, there is one major challenge when applying source code rewriting: one needs to carefully preserve the storage layout. Otherwise, the patched contract will corrupt its memory and fail or (worse) introduce dangerous bugs. Namely, some changes in the source code can break the contract compatibility, even though the changes do not break the logic of the contract.

Statically-sized variables are laid out contiguously in storage starting from address 0. Contiguous variables with size less than 32 byte can be packed into a single 32 byte storage slot [Sol]. As a result, any changes to re-order, add, or remove variables in the source code may look harmless, but on the EVM level, such changes will lead to

a mapping of variables to wrong and unexpected storage addresses. In other words, changes in variable declaration corrupt the internal state of the contract, as the legacy contract and the patched contract have different storage layouts. To mitigate this issue, tools such as Slither [SliU] provide *upgradeability checks*. They assist a developer in checking for contract compatibility, but they do not automatically generate storage-compatible patched contracts.

In EVMPatch, we opted to utilize *bytecode rewriting* to introduce patches to smart contracts. Using bytecode rewriting, we can introduce minimally intrusive patches to the smart contracts. We can ensure that these minimally intrusive patches do not change the storage layout by construction. At the same time, bytecode rewriting is sufficiently powerful to patch many bug classes that only require changes on the level of EVM instructions. Another reason to opt for bytecode rewriting are existing smart contract vulnerability detection tools. As of now, the majority of them operate on the EVM level [FSS18; KR18b; Luu+16; Mos+19] and also report their findings on the EVM level. A bytecode rewriting approach can exploit the reports of these analysis tools to directly generate an EVM bytecode-based patch. Finally, if source-code rewriting is utilized, the developer has limited possibilities to perform thorough testing on the effectiveness of the patched contract. In particular, checking the patched contract against old transactions (including transactions that encapsulate attacks) is easier with bytecode-level patches. Furthermore, analyzing transactions would still require analysis on the bytecode level to reverse-engineer the attack transactions and how they fail against the patched contract. Bytecode-rewriting allows developers to directly match the rewritten bytecode instructions to the attack transactions making forensic analysis feasible. Given all these reasons, we decided to opt for bytecode rewriting.

7.2.2 Framework Design

Our framework depicted in Figure 7.2 consists of the following major components:

1. the *vulnerability detection* engine consisting of automatic analysis tools and public vulnerability disclosures,
2. *bytecode rewriter* to apply the patch to the contract,
3. the *patch testing* mechanism to validate the patch on previous transactions, and
4. the *contract deployment* component to upload the patched version of the contract.

At first, the vulnerability detection adapter component identifies the location and type of the vulnerability using external analysis tools, such as those described in Chapter 5 or Chapter 6. This information is then passed to the bytecode rewriter, which patches the contract according to previously defined patch templates. The patched contract is thereafter forwarded to the patch tester, which replays all past transactions to the contract. That said, we do not only patch the contract, but we allow the developer to retrieve a list of transactions that exhibit different behavior and outcome between the original and patched contract. These transactions serve as an indicator of potential attacks on the original contract. If the list is empty, our framework automatically deploys the patched contract instantly on the Ethereum blockchain. Next, we will provide a more detailed description of the four major components of our design.

Vulnerability Detection Before being able to apply patches, EVMPatch needs to identify and detect vulnerabilities. EVMPatch leverages existing vulnerability detection tools such as [FSS18; FAH20; Gro+18; KR18b; Luu+16; Nik+18; Rod+19; Tsa+18]. For vulnerabilities that are not detected by any existing tool, we require that a developer or a security consultant creates a vulnerability report. In our system, the vulnerability detection component is responsible to identify the exact address of the instruction, where the vulnerability is located, and the type of vulnerability. This information is then passed to the bytecode rewriter, which patches the contract accordingly.

Bytecode Rewriter At the core of our framework is the bytecode rewriter, which applies patches to the smart contract. Binary rewriting is a well-known technique to instrument programs after compilation. Binary rewriting has also been applied to retrofit security hardening techniques such as *control-flow integrity*, to compiled binaries [Dav+12], but also to dynamically apply security patches to running programs [PBG+13]. Two flavors of approaches have been developed for binary rewriting on traditional architectures: static and dynamic rewriting. Dynamic approaches [Luk+05] rewrite code on the fly, i.e., while the code is executing. This avoids imprecise static analysis on large binaries. However, dynamic binary rewriting requires an intermediate layer, which analyzes and rewrites code at runtime. Since the EVM does not support dynamic code generation or modification, it is not possible to apply this approach efficiently in Ethereum.

In contrast, static binary rewriting [BH00; Lau+10] is applicable to Ethereum as it does not require any additional support code at runtime. It relies on static analysis to recover enough program information to accurately rewrite the code. In general, static binary rewriting techniques are well suited for applying patches in Ethereum since smart contracts have comparably small code sizes: typically in the range of about 10 kbyte. The Ethereum main network even enforces a hard limit of 24 kbyte on contract size [But16]. Furthermore, EVM smart contracts are always statically linked to all library code. It is not possible for a contract to dynamically introduce new code into the code address space. This makes the reliance on binary rewriting techniques simpler compared to traditional architectures. As such, smart contracts are a good target for static binary rewriting as many problems of static binary rewriters on classical architectures are avoided due to the nature of EVM smart contracts.

The stack-based architecture of the EVM requires special attention when implementing a patch: all address-based references to any code or data in the code address space of the smart contract must be either preserved or updated when new code is inserted into the code address space. Such references cannot be easily recovered from the bytecode. To tackle this challenge, EVMPatch utilizes a trampoline-based approach for adding new EVM instructions into empty code areas. We describe this trampoline-based approach in more detail in Section 7.3.

To implement a patch, the bytecode rewriter processes the bytecode of the vulnerable contract as well as the vulnerability report. The rewriting is based on a so-called patch template which is selected according to the vulnerability type and adjusted to work with the given contract. Even though a template-based patching approach does not

```
# load owner
PUSHN $owner_address
SLOAD
# mask loaded value, since addresses are 20 bytes long
PUSH20 0xffffffffffffffffffffffffffffffffffffffff
AND
CALLER
EQ
# if (caller() == sload($owner_address))
PUSHL $CONTINUE
JUMPI
# revert(0, 0)
PUSH1 0x00
DUP1
REVERT
JUMPDEST CONTINUE
```

Figure 7.3: Patch template for the *only owner* patch template that implements an access control check. EVMPatch specializes this template according to the storage address of the *owner* variable. Furthermore, the assembler built into EVMPatch can handle several pseudo-opcodes such as `PUSHN` to push constants with unknown size and `PUSHL`, which pushes the address of following `JUMPDEST` instruction. i.e., it performs relocation of the code at the time of rewriting.

offer the same flexibility as source-level patching, it allows us to cover many common types of vulnerabilities.

Patch Templates In EVMPatch, we utilize a template-based patching approach: for every supported class of vulnerabilities, a patch template is integrated into EVMPatch. This patch template is automatically adapted to the contract that is being patched. We create generic patch templates such that they can be easily applied to all contracts. EVMPatch *automatically* adapts the patch template to the contract at hand by replacing contract-specific constants (i.e., code addresses, function identifiers, storage addresses). Patch templates for common vulnerabilities, such as integer overflows, are shipped as part of EVMPatch, and a typical user of EVMPatch will never interact with the patch templates. However, optionally, a smart contract developer can also inspect or adapt existing patch templates or even create additional patch templates for vulnerabilities that are not yet supported by EVMPatch. EVMPatch supports two formats for custom patch templates. The first is based on EVM instructions as shown in Figure 7.3, which are assembled and relocated during bytecode rewriting. EVMPatch also supports a *yaml* based configuration file that describes patches to a contract. An example of this format is depicted in Figure 7.4. Here a developer can utilize simple domain-specific expression language that resembles Python expressions to describe patches that enforce pre-conditions on functions similar to Solidity modifiers.

Patch Tester As smart contracts directly handle assets (such as Ether or Tokens), it is critical that any patching process does not impede the actual functionality of a contract. As such, any patch must be tested thoroughly. To address this issue, we introduce a patch-testing mechanism, which is based 1) on the transaction history recorded on the blockchain and 2) optional developer-supplied unit tests. At this point, we exploit the fact that any blockchain system records all previous executions of a smart contract, i.e., transactions in Ethereum. In our case, the patch tester re-executes all existing transactions and optionally any available unit test and verifies that all transactions of the old legacy and the newly patched contract behave consistently. The patch tester detects any behavioral discrepancy between the old legacy and the newly patched contract and reports a list of transactions with differing behavior to the developer. That said, as a by-product, our patch-testing mechanism can be used as a forensic attack detection tool. Namely, while executing the patching process, the developer will also be notified of any prior attacks that abuse any of the patched vulnerabilities and can then act accordingly. In case both versions of the contract behave the same way, the patched contract can be automatically deployed. Otherwise, the developer must investigate the list of suspicious transactions and thereafter invoke the contract deployment component to upload the patched contract. The list of suspicious transactions may not only serve as an indicator of potential attacks but may reveal that the patched contract is not functionally correct, i.e., the patched contract shows a different behavior on benign transactions. In Section 7.4, we provide a thorough investigation of real-world, vulnerable contracts to demonstrate that EVMPatch successfully applies patches without breaking the original functionality of the contract.

Contract Deployment As discussed in Section 7.1.1, the delegatecall-proxy-based upgrade scheme is the option of choice to enable almost instant contract patching. Thus, EVMPatch integrates this deployment approach by utilizing a proxy contract as the primary entry point for all transactions with a constant address. Before the first deployment, EVMPatch transforms the original unmodified contract code to utilize the delegatecall-proxy pattern. This is done by deploying a proxy contract, which is immutable and assumed to be implemented correctly. For example, EVMPatch comes with a default proxy contract that is only 80 lines of Solidity code. However, it is trivial to integrate other proxy contracts, such as those with recent standardization efforts [IA; MWM]. The original bytecode is then converted to a logic contract using the bytecode rewriter with only minor changes to the original code. The logic contract is then deployed alongside the proxy contract.

Patch Deployment Finally, when the contract is patched and after the patch is tested by the patch tester component, EVMPatch can deploy the newly patched contract. Our upgrade scheme deploys the newly patched contract code to a new address. It then issues a dedicated transaction to the previously deployed proxy contract, which switches the address of the logic contract from the old vulnerable version to the newly patched version. The patched logic contract now handles any further transactions.

```
add_require_patch:
  deposit:
    - "sload(owner) != 0"
  withdraw: []
  migrateTo:
    - "sload(0) == caller()"
```

Figure 7.4: Patch configuration file that describes a patch to EVMPatch and is used for manual patching assisted by EVMPatch. Here two patches are added to the contract. The first adds a pre-condition check to the *deposit* function, which prevents calling this function if no owner is set. The second patch restricts access to the *migrateTo* to the owner, which is stored at storage address 0.

Human Intervention We designed EVMPatch to be fully automated. However, there are a few scenarios where developer intervention is required. Namely, if (1) the vulnerability report relates to a bug class that is not yet supported by EVMPatch, or (2) the patch tester reports at least one transaction that fails due to the newly introduced patch and the failing transaction is not a known attack transaction, (3) the patch tester reports that at least one known attack transaction is not prevented by the newly introduced patch.

If a bug class is not supported, EVMPatch informs the developer about the unsupported vulnerability class. Since EVMPatch is extensible, it easily allows developers to provide custom patch templates thereby allowing quick adaption to new attacks against smart contracts.

If the patch tester finds a new failing transaction, the developer has to analyze whether a new attack transaction has been discovered or a legitimate transaction has failed. For a newly discovered attack transaction, EVMPatch adds this transaction to the list of attacks and proceeds. Otherwise, the developer investigates why the legitimate transaction failed. As our evaluation in Section 7.4 shows, such cases typically occur due to inaccurate vulnerability reports, i.e., wrongly reported vulnerabilities rather than faulty patching. Thus, the developer can simply blacklist the wrongly reported vulnerable code locations to avoid patching at these locations.

These manual interventions typically only need quick code reviews or debugging sessions. We believe even moderately experienced Solidity developers can perform these tasks as no detailed knowledge about the underlying bytecode rewriting system is needed. As such, EVMPatch positions itself as a tool to enable more developers to securely program and operate Ethereum smart contracts.

If the patch tester reports that the patch does not prevent a known attack transaction, then the patch is not effective. In this case, EVMPatch notifies the developer and must investigate whether the vulnerability report accurately describes the actual exploitation attempt. Then the developer must manually refine the vulnerability report to include the accurate vulnerability class and location such that EVMPatch can apply the right patch template at the right place.

7.3 Implementation of EVMPatch

In this section, we describe the implementation of EVMPatch according to the architecture presented in Section 7.2. Previously, in Section 7.1.2, we discuss engineering challenges for bytecode rewriting in Ethereum. In the following chapter, we describe how we tackle these challenges in EVMPatch. We describe the implementation of the bytecode rewriter (Section 7.3.1), the patch testing feature (Section 7.3.2), and the contract deployment mechanism (Section 7.3.3).

7.3.1 Trampoline-based Bytecode Rewriting

We implement a trampoline-based rewriter in Python and utilize the *pyevmasm*[1] library for disassembling and assembling raw EVM opcodes. Our trampoline-based bytecode rewriter works on the basic block level and utilizes an algorithm similar to the algorithm shown in Algorithm 1 to identify basic block boundaries. In contrast to the algorithm described in Algorithm 1, we apply the analysis only to the relevant parts of the code.

When an instruction needs to be instrumented, the whole basic block that contains the instruction is copied to the end of the contract. The bytecode rewriter then applies the patch to this new copy of the basic block. The original basic block is replaced with a trampoline, i.e., a short instruction sequence that immediately jumps to the copied basic block. Whenever the contract jumps to the basic block at its original address, the trampoline is invoked, redirecting execution to the patched basic block by means of a jump instruction. To resume execution, the final instruction of the instrumented basic block issues a jump back into the original contract code. Figure 7.5 shows an example of the bytecode rewriting process. Here the basic block at `0xAB` is replaced by a trampoline, and the original basic block is moved to the address `0xFFB`. While the trampoline-based approach avoids fixing up any references, it introduces additional jump instructions. However, as we will show, the gas cost associated with these additional jumps is negligible in practice (see Section 7.4).

With the trampoline approach, all addresses that reference the original contracts bytecode stay the same. This means our rewriter must not fix up any references and, as such, avoids the need for complex and failure-prone data-flow analysis. However, to ensure correct execution, we must still compute a partial CFG, starting from the patched basic blocks. This is necessary to recover the boundaries of the basic blocks that are patched and the following basic blocks that are connected by a so-called fall-through edge. Not all basic blocks terminate with an explicit control-flow instruction: Whenever a basic block ends with a conditional jump instruction (`JUMPI`) or simply does not end with a control-flow instruction, there is an implicit edge (i.e., fall-through) in the control-flow graph to the instruction at the following address.

Handling Fall-Through Edge Two cases must be considered when handling the fall-through edge. When the basic block targeted by the fall-through edge starts with a `JUMPDEST` instruction, the basic block is marked as a legitimate target for regular jumps in the EVM. In this case, we can append an explicit jump to the rewritten basic

[1] github.com/crytic/pyevmasm

block at the end of the contract and ensure that execution continues at the beginning of the following basic block in the original contract code. In case the following basic block does not begin with a `JUMPDEST` instruction, the EVM forbids explicit jumps to this address. In the CFG, this means that this basic block can only be reached with a fall-through edge. To handle this case, our rewriter copies the basic block to the end of the contract right behind the rewritten basic block constructing another fall-through edge in the CFG of the rewritten code.

Figure 7.5 shows an example of how our rewriter changes the CFG of the original contract. The `ADD` instruction is replaced with a checked add routine that additionally performs integer overflow checks. We call the address of the `ADD` instruction the *patch point*. The basic block, which contains the patch point, is replaced with a trampoline. In this case, it immediately jumps to the basic block at `0xFFB`. This basic block, which is placed at the end of the original contract, is a copy of the original basic block at `0xAB`, but with the patch applied. Since the basic block is now at the end of the contract, the bytecode rewriter can insert, change, and remove instructions in the basic block without changing any address in the code that is located at higher-numbered addresses. We fill the rest of the original basic block with the `INVALID` instruction to ensure the basic block has the exact same size as the original basic block. The basic block at `0xCD` is connected to the prior basic block by means of a fall-through edge. However, this basic block starts with a `JUMPDEST` instruction and, as such, is a legitimate jump target. Hence, the rewriter then appends a jump to the patched basic block at `0xFFB`, which ensures execution continues in the original contract's code at address `0xCD`.

Adapting to Solidity Smart Contracts The EVM has some particularities that must be considered when implementing a bytecode rewriter. As discussed in Section 3.2.4 the EVM enforces some separation of code and data in the code address space. Namely, the EVM disallows jumps into the data constants that are embedded into `PUSH` instructions. However, such push instructions are often part of constant data, which is accumulated at addresses strictly larger than any reachable code by smart contract compilers. This avoids any conflicts between the generated code and data encoded into the code address space. However, our trampoline-based rewriter does append code behind the data constants of the smart contracts. When the rewriter appends code, it avoids that the new code is accidentally marked as an invalid jump destination due to a preceding push opcode byte. The rewriter avoids this by carefully inserting padding between the data of the original contract and the newly appended code. To compute the necessary length of the padding, we leverage the common EVM linear sweep algorithm, as described in Section 3.2.4 to determine the first valid code address after the end of the original contract.

Furthermore, most contracts contain additional meta-data at the end of the contract. For instance, Solidity appends a special encoding of a hash value, called the swarm hash, at the end of each contract [SolSH]. It is expected that a compiled Solidity contract ends with the swarm hash. As such, our bytecode rewriter handles the swarm hash in a special way. Instead of appending code after the swarm hash, the bytecode rewriter inserts new code and data right before the original swarm hash. This ensures

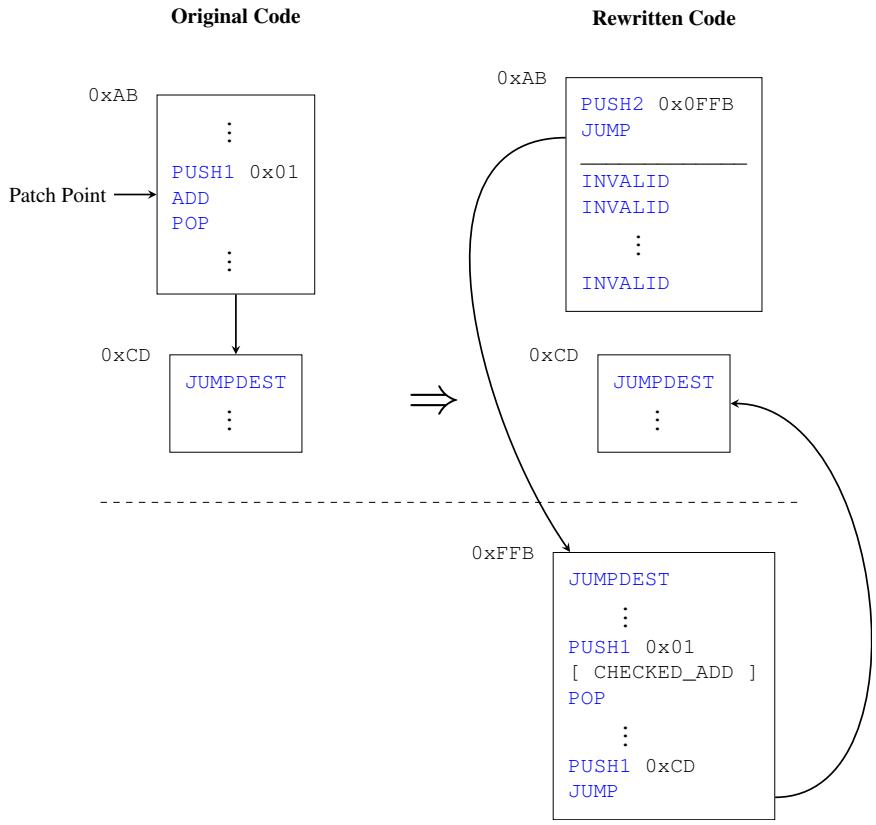


Figure 7.5: Control-flow graph of original and rewritten code. The end of the original contract's code is marked with a dashed line. The original basic block at address `0xAB` is replaced with trampoline code that jumps to the patched version of the basic block, located at address `0xFFB`. The patched basic block is appended right after the end of the original contract at address `0xFFB`. The patched basic block jumps back into the original code at address `0xCD`.

that the swarm hash is always at the end of the contract, even after rewriting. Normal contracts will not reference the swarm hash in their code and as such, the address of the swarm hash can be changed without impacting the original code.

Applicability of Trampoline Approach The trampoline-based approach to rewriting requires only minimal code analysis and works for most use cases. However, this approach faces two problems. First, instructions can only be patched in basic blocks that are large enough (in terms of size in bytes) to also contain the trampoline code. However, a typical trampoline requires 4 to 5 bytes, and typically, basic blocks that perform some meaningful computation are large enough to contain the trampoline code. Second, due to the copying of basic blocks, the code size increases depending on the patched block, thereby increasing deployment cost. However, our experiments show that the overhead during deployment is negligible (on average US$ 0.02 per deployment, see Section 7.4).

No reliance on accurate CFG Recovering an accurate CFG given only EVM bytecode is a challenging problem. However, our trampoline-based approach does not require an accurate and complete CFG. Instead, we only need to recover basic block boundaries given the program counter of the instruction, where the patch needs to be applied. In doing so, recovering the basic block boundaries is tractable since the EVM has an explicit marker for basic block entries (see Section 3.2.4). Furthermore, our rewriter only needs to recover the end of the basic block and any following basic blocks that are connected via fall-through edges in the CFG.

7.3.2 Patch Testing

While the insertion of trampolines into the original code does not change the functionality of the contract, the patch template itself can perform arbitrary computations and could potentially violate the semantics of the patched contract. To test the patched contract, EVMPatch utilizes a differential testing approach. That is, we re-execute all transactions of the contract to determine if the behavior of the original, vulnerable code and the newly patched code differ. EVMPatch utilizes past transactions to the contract retrieved directly from the blockchain. If the contract comes with unit tests, EVMPatch can also utilize the unit tests to test the newly patched contract. While this differential testing approach cannot guarantee formal correctness of the contract, it raises confidence in the patched contract.

One potential problem for any testing-based approach is a low test coverage. In the history-based testing, which we utilize in EVMPatch, contracts with a low number of available transactions are also prone to low test coverage. However, our experiments (see Section 7.4) show that the differential testing approach works well enough in practice to show that the patches do not break functionality. Given the availability of a formal specification of the contract's functionality, EVMPatch could also leverage a model checker to validate a patched contract more rigorously.

During differential testing, we first retrieve a list of transactions to the vulnerable contract from the blockchain. Second, we re-execute all those transactions and retrieve the execution trace for each transaction. Then, we then re-execute the same transactions but replace the code of the vulnerable contract with the patched contract code to obtain the second execution trace. We use a modified Ethereum client, based on the popular *go-ethereum* client[2] since the original client does not support this functionality. Finally, we compare both execution traces, and the patch tester produces a list of transactions where the behavior differs. If there are no such transactions, then we assume that the patch does not inhibit the functionality of the contract and proceed with deploying the patched contract.

The execution traces of the original and patched contracts are never equal since patching changes control flow and inserts instructions. Hence, we examine only potentially state-changing instructions, i.e., instructions that either write to the storage area (i.e., a `SSTORE`) or transfer execution flow to another contract (e.g., a `CALL` instruction). We then compare the order, parameters, and result of all state-changing instructions and find the first instruction where the two execution traces

[2]We utilized version 1.8.27-stable-3e76a291

differ. Currently, we assume that the introduced patches do not result in any new state-changing instructions. This assumption holds for patches that introduce input-validation code and revert when invalid input is passed. However, the trace difference computation can be adapted to become aware of potential state changes that a patch introduces.Reported transactions that fail in the code, which is part of the patch, are marked as potential attack transactions. If the reported transaction failed due to out-of-gas in the patched code, we rerun the same transaction with an increased gas budget. We issue a warning since users will have to account for additional gas costs introduced by the patch. Finally, the developer must examine the reported transactions to decide whether the given list of transactions is legitimate or malicious. As a side-effect, this makes our patch tester an attack detection tool for the vulnerable contract allowing developers to quickly find prior attack transactions.

7.3.3 Deployment of Patched Contracts

As described in Section 7.2, EVMPatch utilizes the delegatecall-proxy based upgrade pattern to deploy the patched contract. To achieve this, EVMPatch splits the smart contract to two contracts: a proxy contract and a logic contract. The proxy contract is the primary entry point and stores all data. By default, EVMPatch utilizes a proxy contract that is shipped with EVMPatch. However, EVMPatch can also re-use existing upgradable contracts, such as contracts developed with the ZeppelinOS framework [ZepOS]. Users interact with the proxy contract, which is located at a fixed address. To facilitate the upgrade process, the proxy contract also implements functionality to update the address of the logic contract. To prevent malicious upgrades, the proxy contract also stores the address of an owner who is allowed to issue upgrades. The upgrade then simply consists of sending one transaction to the proxy contract, which will (1) check whether the caller is the owner and (2) update the address of the logic contract.

The proxy contract retrieves the address of the new logic contract from storage and simply forwards all calls to that contract. Internally, the proxy contract utilizes the `DELEGATECALL` instruction to call into the logic contract. This allows the logic contract to gain full access to the proxy's storage memory area, thereby allowing access to the persistent data without any additional overhead.

7.3.4 Application to Vulnerability Classes

The bytecode rewriter takes a patch template, which is specified as a short snippet of EVM assembly language. This template is then specialized according to the patched contract and relocated to the end of the patched contract. This template-based approach to patch generation allows specifying multiple generic patches to address whole classes of vulnerabilities, such as 1) mishandled exceptions [Luu+16], 2) reentrancy (see Section 3.2.3), 3) access control issues [KR18a; Nik+18], and 4) integer bugs [FSS18]. In the following, we discuss two vulnerability classes that can immediately benefit from our framework.

```
function initMultiowned(address[] _owners, uint _required)
                      ①   internal {
// ...
function initDaylimit(uint _limit) ①   internal {
// ...
// throw unless the contract is not yet initialized.
modifier only_uninitialized { if (m_numOwners > 0) throw; _;}

function initWallet(address[] _owners, uint _required,
                 uint _daylimit)
                 ②   only_uninitialized {
// ...
```

Figure 7.6: Source code of patched Parity Multisig Wallet.

Improper access control

Improper access control to critical functions can be patched by just inserting a check at the beginning of a function to verify that the caller is a certain fixed address or equal to some address stored in the contract's state. Prior work [KR18b; Nik+18] investigated detection tools to handle this vulnerability. We can also utilize EF$\flat$CF, introduced in Chapter 5, to identify access control issues, such as unprotected selfdestructs.

The Parity MultiSig Wallet is a prominent example of access control errors [Bre+17; Tec17b]. This contract implements a wallet that is owned by multiple accounts. Any action taken by the wallet contract must be authorized by at least one of the owners. However, the contract suffered from a fatal bug that allowed anyone to become the sole owner because the corresponding functions *initWallet*, *initMultiowned*, and *initDayLimit* did not perform any access control checks.

Figure 7.6 shows the patched source code which adds the *internal* modifier to the functions *initMultiowned* and *initDayLimit* (marked with ① in Figure 7.6). This modifier makes these two functions inaccessible via the outside interface of the deployed contract. Furthermore, the patch adds the custom modifier *only_uninitialized*, which checks whether the contract was previously initialized (marked with ②).

The developers originally introduced a new vulnerability while deploying the patched contract, which was actively exploited [Tec17a]. In contrast, because EVMPatch performs bytecode rewriting, it would have immediately generated a securely patched version of the contract and would have deployed it automatically in a secure manner.

Consider Figure 7.7, which shows a customized patch in the domain-specific language employed by EVMPatch to specify patches. As such, we insert a patch at the beginning of the *initWallet* function that checks whether the condition `sload(m_numOwners) == 0` holds, i.e., whether the contract is not yet initialized. If this does not hold, the contract execution will abort with a `REVERT` instruction. Note that in the patch in Figure 7.7, we require an explicit `sload` to load variables from storage. Further, the expression is logically inverted compared to the source-level patch since this patch essentially inserts a Solidity `require` statement. Furthermore, two other publicly accessible functions need to be removed from the public function dispatcher. The patch shown in Figure 7.7 combines two existing patch templates provided by EVMPatch. First, the *add require*

```
add_require_patch:
  initWallet:
    - sload(m_numOwner) == 0

delete_public_function_patch:
  - initDayLimit
  - initMultiowned
```

Figure 7.7: Customized Patch for Partity Multsig Wallet.

patch template enforces a pre-condition before a function is entered. Second, the *delete public function* patch template removes a public function from the dispatcher, effectively marking the function as internal.

We verified that both the source-level and EVMPatch-ed contracts successfully prevent an attack. This shows the potential of EVMPatch to patch a wide variety of different bugs, including access control bugs. In this example, we manually constructed the access control patch. Due to the application-specific nature of access control bugs, it is highly challenging to deduce a correct patch for fixing access control bugs. The information given by existing analysis tools [FAH20; KR18b; Nik+18] is not adequate to automatically construct a patch, i.e., they only give a vulnerable execution trace and not the root cause of the vulnerability. As such, we leave the fully automatic patching of access control patches as future work. However, in the following, we describe how to use a vulnerability detection component to fully automate the patching process with EVMPatch.

Integer bugs

Integer bugs are highly likely to occur when dealing with integer arithmetic since Solidity does not utilize checked arithmetic by default. This has resulted in many potentially vulnerable contracts being deployed and some being actively attacked [FSS18; Pecc]. Given the prevalence of these vulnerabilities, we discuss in the remainder of this section how to automatically patch integer overflow bugs using EVMPatch.

Typical integer types are bounded by a minimum and/or maximum size due to the fixed bit-width of the integer type. However, programmers often do not pay sufficient attention to the size limitation of the actual integer type, potentially causing integer bugs. Several high-level programming languages (Python, Scheme) are able to avoid integer bugs since they leverage arbitrary precision integers with virtually unlimited size. Other programming languages feature explicitly checked arithmetic (e.g., Rust). Only recently, in version 0.8.0, Solidity started to switch to checked integer arithmetic [Sol080] by default. However, many smart contracts still utilize older Solidity versions, which did not feature any safeguard against integer bugs at all. This left the burden of handling integer overflows completely on the developer, who needs to either manually implement overflow checks or properly utilize the SafeMath library to safely perform numeric operations [OZSM]. While common, the former is error-prone, and the latter needs to be applied rigorously.

```
1  function batchTransfer(address[] _receivers, uint256 _value)
2                         public whenNotPaused returns (bool) {
3    uint cnt = _receivers.length;
4  // OVERFLOW: 2 * ((INT_MAX / 2) + 1) == 0
5    uint256 amount = uint256(cnt) * _value;
6    require(cnt > 0 && cnt <= 20);
7  // BYPASSED CHECK: balances[msg.sender] >= 0
8    require(_value > 0 && balances[msg.sender] >= amount);
9  // RESULT: Transfer of ((INT_MAX / 2) + 1) tokens
```

Figure 7.8: Integer overflow bug reported by PeckShield [Peca].

For instance, multiple vulnerabilities in ERC-20 token contracts were unveiled [Pec18; Peca; Pecb]. These contracts manage subcurrencies, so-called tokens, on the Ethereum blockchain. Such tokens can deal with large amounts of currency since they track the token balance of every token owner and mediate the exchange of tokens and Ether. Figure 7.8 shows an excerpt of the *BEC* token contract's code as an example of such an integer overflow vulnerability. When computing the total amount in Line 6, the contract uses an unchecked integer multiplication. If an attacker provides a very large *_value*, the overflow will be triggered, and as a consequence, the *amount* variable will be set to a small amount. This effectively bypasses the balance check in Line 11, which verifies that the balance of the attacker is high enough to transfer the requested tokens as specified in the *_value* parameter. This allows the attacker to transfer many tokens to an attacker-controlled account. Subsequently, the attacker can then attempt to exchange the tokens for Ether or other currency.

We developed patch templates for detecting integer overflows and underflows for the standard EVM integer width, i.e., unsigned 256 bit integers. For integer addition, subtraction, and multiplication, these templates add checks inspired by secure coding rules in the C programming language [Rob+] and the *SafeMath* [OZSM] Solidity library. When a violation is detected, EVMPatch issues an exception to abort and roll back the current call to the contract.

The patch templates replace a single arithmetic instruction, which is vulnerable to integer overflows, with a checked variant. As such, our patches are optimized for minimal gas overhead and minimal intrusiveness into the contract.

Call-Related Patches

In the following, we discuss the capability of EVMPatch in patching call-related vulnerabilities: reentrancy and mishandled exceptions. We have not evaluated these types of patches, but they are straightforward to implement as they focus on external calls, which are all implemented with the `CALL` instruction in EVM. The patch point naturally becomes the `CALL` instruction. Both types of patches can be combined. In fact, both types of patches can also be applied without using any prior analysis tool to harden any external call at the cost of higher runtime overhead.

Reentrancy To protect against reentrancy attacks, EVMPatch can utilize an approach similar to the popular *REGuard* library provided by the *OpenZeppelin* project [REGu]. Here there is a single reentrancy lock that protects all functions. To adapt this, EVMPatch inserts an `SSTORE` instruction to set the global lock before the `CALL` instruction and another `SSTORE` after the call to release the global lock. To avoid an incompatible storage layout, the patch uses a random storage address for the global lock, which is selected using a cryptographically secure random number generator when preparing the patch. The chance of collisions with other storage addresses is quite low due to the vast space of storage slots (2^{256} storage slots are available). Finally, EVMPatch inserts a check that makes sure the global lock is not set at the very beginning of the first basic block of the contract. This way, EVMPatch comprehensively protects all functions of the smart contract. Alternatively, EVMPatch could insert this check only at the beginning of selected functions using developer annotations or guidance by a vulnerability detection tool such as EF/CF (see Chapter 5). The downside of this approach to patching reentrancy is that it does not allow for more fine-grained protection of storage variables, which would require more complex data-flow analysis as discussed in Chapter 6.

Mishandled Exceptions This type of bug is also quite easy to patch using EVMPatch. Typically, the problem with mishandled exceptions is a lack of checking the return value of external calls. Even if the external call fails and the whole sub-transaction triggered by the external call is reverted, the buggy smart contract will continue with execution. To patch this, EVMPatch inserts a generic check for the success of the external call right after the `CALL` instruction. If the external call does not succeed, the patch will trigger a *revert* of the current transaction. However, such a generic patch has the downside that it does not allow a developer to intentionally ignore the state of an external call. Furthermore, in current Solidity versions, the low-level call operations that require manual exception handling have been discouraged. The Solidity compiler now warns the developer if a call returning due to an exception is not handled. As such, this type of vulnerability has become less relevant since the early discussions of Solidity security issues [ABC17; Luu+16].

7.4 Evaluation

We perform an extensive evaluation using our EVMPatch prototype. First, we show that EVMPatch is capable of patching access control bugs using manually written patch specifications, based on an example from Section 7.3.4. In the remainder of the evaluation we focus on our fully automated patching pipeline for integer bugs, which we implemented with the help of the Osiris [FSS18] tool. We utilize our patch testing component to evaluate (1) the correctness of the rewriter, (2) the gas overhead, (3) the code size increase, and (4) the increased deployment costs. Furthermore, we take a detailed look at the timeline of the attack transactions identified by EVMPatch on several ERC-20 token contracts. We conclude with an analysis of false alarms and missed bugs in the Osiris tool, which directly affect also EVMPatch's precision.

Running Example: Access Control

In Section 7.3.4 we discussed how to apply EVMPatch to semi-automatically patch an access control bug in the prominent *Parity Multisig Wallet* contract. We compare a manual source-level patch with the patch applied by EVMPatch. We verified that both the EVMPatch'ed and the manually patched contracts are no longer exploitable. We deploy the patched versions of the *WalletLibrary* contract and perform the attack, which now fails due to the patches. We then compare both the manual and the EVMPatch'ed versions in terms of their gas overhead and code size increase. Table 7.1 shows an overview of the results.

EVMPatch only increases contract size by 25 byte. The additional gas cost of the *initWallet* function is only 235 gas, i.e., 0.000 06 USD per transaction for 235.091 USD/ETH and a typical gas price of 1 Gwei. In practice, the gas increase induced by EVMPatch is negligible. This demonstrates that EVMPatch can efficiently and effectively insert patches for access control bugs. However, in contrast to bytecode rewriting, the source-level patch removes public functions much more efficiently. As such, they can also reduce the size of the final contract bytecode.

Rewriter Correctness

To verify the correctness of the patches generated by our bytecode rewriter, we utilized the state-of-the-art integer detection tool Osiris [FSS18] for vulnerability detection. After analyzing 50 535 unique contracts in the first 5 000 000 blocks of the Ethereum blockchain, Osiris detects at least one integer overflow vulnerability in 14 107 contracts. Using EVMPatch, we were able to successfully patch almost all of these contracts automatically. More specifically, we could not patch 33 contracts amongst the 14 107 investigated contracts because the basic block, where the detected vulnerability was located is too small for the trampoline code.

From those 14 107 contracts, around 8000 involve transactions on the Ethereum network. We execute the patch tester over these contracts to detect potential attacks and test the correctness of our patch. To generate a large and representative evaluation data set, we extracted all transactions sent to these contracts up to block 7 755 100 (May 13th, 2019) from the Ethereum blockchain resulting in 26 385 532 transactions.

Replaying those transactions with our patch tester shows that for 95.5 % of all vulnerable contracts, EVMPatch's generated patch was compliant with all of the prior transactions associated with those contracts. For the remaining 4.5 % of the investigated contracts, our patch rejected transactions for one of the following reasons:

Table 7.1: Overhead of access control patch.

Version	Code Size (byte)	Size Increase	Gas Increase
Original	8290	0 %	0
Source-Patched	8201	−1.07 %	226
EVMPatch'ed	8315	0.3 %	235

(1) we successfully stopped a malicious transaction, (2) the reported vulnerability was a false positive and should not have been patched, or (3) we unintentionally changed the contract's functionality. Due to the high reverse engineering efforts required for analyzing and debugging the bytecode of the remaining 4.5 %, we do not provide a comprehensive analysis.

For close scrutiny, we selected five ERC-20 token contracts with confirmed integer overflow/underflow vulnerabilities that have been successfully attacked (see Table 7.2). We utilize these token contracts to compare EVMPatch with other patching methods, notably manual source-level patching. For comparison purposes, we also manually patch these contracts on the Solidity source code level by replacing the vulnerable arithmetic operations with functions adapted from the *SafeMath* library [OZSM]. The manually patched source code is then compiled with the exact same Solidity compiler version and optimization options used in the original contract (as reported on etherscan.io). We provide the results of our comparison in Table 7.3.

We applied the EVMPatch patch tester to the generated patched contract versions and validated the reported outcome. This allows us to verify whether both patching approaches abort the same attack transactions. In addition, we can compare the overhead in gas consumption and the increase in code size. Note that in the manual patching method, we do not patch all potential vulnerabilities detected by Osiris. To simulate realistic patching, we do not add checks on those arithmetic operations which cannot be exploited by an attacker. More specifically, we do not patch vulnerable arithmetic operations contained in functions that can only be called by the controller or owner of the contract. We verified the correctness of our patches using a total number of 506 607 real-world transactions associated with the ERC-20 token contracts listed in Table 7.2.

Table 7.3 shows the transaction execution results of the patch tester. We verified the aborted transactions and confirmed that all of them correspond to genuine attacks except for one transaction[3], which resembles a special case of token burning that we discuss in detail below. Apart from the valid attack transactions, the execution traces of the re-executed transactions match those of the original transactions, confirming that our patch does not break the contract's functionality.

Out of the transactions identified as attacks, we found one particular transaction to the HXG token [HXG]. The transaction does indeed trigger an integer overflow. However, the HXG contract burns some tokens by transferring them to a blackhole address `0x0`. The burned tokens cannot be recovered, and the balance of the blackhole address does not influence the behavior of the contract. When analyzing the contract, Osiris is not aware of the semantics of this blackhole address and reports a possible integer overflow. EVMPatch then conservatively patches the integer overflow bugs reported by Osiris, which leads to one legitimate transaction failing. We argue that this pattern can be seen as bad coding practice as it wastes gas by unnecessarily storing the balance of the blackhole address. We discuss this particular case along with other false alarms in more detail in Section 7.4.

[3] 0x776da02ce8ce3cc882eb7f8104c31414f9fc756405745690bcf8df21e779e8a4

Gas Overhead

The additional code introduced by the patching may potentially cause transactions to fail with an out-of-gas error. While the patches generally do not significantly increase gas consumption, such a behavior can nevertheless occur when the sender of the transactions provides a very tight gas budget. When the re-execution of a transaction with patched code fails early due to an out-of-gas exception, we could not accurately compare the behavior of the patched contract with the original contract. To remedy this, we disabled the gas accounting in the EVM. We report the amount of additional gas consumption during transaction execution in Table 7.3. We excluded those transactions that do not execute functions which contain the vulnerable code. They are not affected by the patches and are, therefore, not relevant to our measurements. As such, neither the bytecode rewriter nor the manual SafeMath patches have added any gas overhead.

Our results show that for contracts BEC, SMT, and HXG, those patched with EVMPatch incur less gas overhead at runtime (83 gas, 47 gas and 120 gas) when compared to those patched on the source code level (164 gas, 108 gas and 541 gas). This is due to the fact that the Solidity compiler generates non-optimal code when only very few checks are added. In particular, Solidity utilizes internal function calls to invoke the SafeMath integer overflow checks. While this reduces code size (in case the check is needed at multiple places), it always requires executing additional instructions—thereby increasing gas overhead—to invoke and return from the internal function. In contrast, EVMPatch inlines the safe numeric operations, thereby introducing less gas overhead. One would need to instruct the Solidity compiler to selectively enable function inlining to yield similar gas costs as EVMPatch.

Note that the average gas overhead is 0 gas for the manually patched SCA token. This is because only one transaction triggers the SafeMath integer overflow check. However, this is an attack transaction, and it is aborted early, making gas overhead calculation not possible.

For UET and SCA, we identify higher gas overhead than for the manually patched version. In fact, UET requires, on average, 255 units of additional gas for every transaction in the patched version. In contrast, only 21 gas is added by the manually patched version. This is due to the fact that our bytecode rewriter conservatively patches every potential vulnerability reported by Osiris in these two contracts (12 and 10, respectively). However, not all of them are exploitable; as such, we did not instrument them during manual patching.

Code Size Increase

Deploying contracts in the Ethereum blockchain also incurs costs proportionally to the size of the deployed contract. More specifically, Ethereum charges 200 gas per byte to store the contract code on the blockchain [Woo19]. From Table 7.3, we recognize that the amount of extra code added by our rewriter is comparable to that of the SafeMath approach when a single vulnerability is patched. Since our approach duplicates the original basic blocks, the code size overhead depends on the specific location of the vulnerability. In the case of the BEC token contract, our rewriter increases the code

size less than the source-level patches. The Solidity compiler generates more code for including the SafeMath library than is strictly necessary for the patch. Even considering the overhead of bytecode rewriting, we observe that EVMPatch generates a smaller patch than the manual patching method for this contract.

However, in case many vulnerabilities are patched, EVMPatch adds a slightly higher overhead. Naturally, the size of the upgraded contracts increases with the number of vulnerabilities to fix due to inlining. For instance, our bytecode rewriter generates 12 patches for the UET contract and ten patches for the SCA contract resulting in 1299 byte (18.2%) and 3811 byte (17.3%) increase in code size. In the worst-case scenario in our dataset, this increase in code size induces negligible additional cost of US$ 0.18 per deployment.

Our patch templates are currently optimized for patching a single vulnerable arithmetic. It is straightforward to adopt an approach akin to Solidity's internal function calls when developing patch templates for our bytecode rewriter, which would reduce the code size overhead when patching many integer overflows.

EVMPatch applies 3.9 patches on average to the contracts in our data set of 14 107 contracts. The average code size of the original contracts is 8142.7 byte ($\sigma = 5327.8$ byte). The average size increase after applying patches with EVMPatch is 455.9 byte ($\sigma = 333.5$ byte). This amounts to an average code size overhead of 5.6% after applying the patches. Given that Ethereum charges 200 gas per byte to the contract creation transaction, it incurs an average overhead of 91 180 gas or US$ 0.02 at the time of writing[4]. In the worst case that we observed, EVMPatch incurs an overhead of 199 800 gas at deployment, which at the time of writing amounts to only about US$ 0.04 additional deployment cost. This shows that the overhead of applying patches with bytecode rewriting is negligible for contract deployment, especially when compared to the number of Ether possibly at stake.

Costs of Deployment

The deployment cost of a newly patched contract dominates the costs of operating a smart contract with EVMPatch. Additionally, there is a transaction needed to switch the address of the logic contract. Since the proxy pattern requires no state migration, this transaction requires a constant amount of gas. The proxy contract we utilize in EVMPatch consumes 43.167 gas during a switchover transaction, i.e., about US$ 0.01. Currently, state migration is the most viable contract upgrade strategy besides the proxy pattern. Prior work estimated that even with only 5000 ERC-20 holders, i.e., smart contract users, state migration will likely cost more than US$ 100.00 in the best case [Tra18]. Hence, compared to the cost of migrating all data to a new contract, the EVMPatch's additional cost of US$ 0.01 is negligible.

[4]The calculation is based on 235.091 USD/ETH and a typical gas price of 1 Gwei

Table 7.2: ERC-20 Token contracts investigated in depth with their respective CVE number, the number of patches introduced by EVMPatch, and the number of transactions replayed by EVMPatch's patch tester and the number of attack transactions identified while testing the patches.

Contract	CVE	# Patches	# Transactions	
			Total	Attacks
[BEC]	2018-10299	1	424 229	1
[SMT]	2018-10376	1	56 555	1
[UET]	2018-10468	55	24 034	12
[SCA]	2018-10706	1	292	10
[HXG]	2018-11239	9	1497	5

Table 7.3: Average amount of overhead in gas consumption over all replayed transactions and overhead of contract size of the manual patched contracts (SM) and rewriter-generated patches (RW) and the overhead of the rewriter converted to US$ (with a gas price of 1 Gwei and 235 US$/Ether; For readability we only show the exact US$ figures only if they are more than one cent).

Contract	Overhead (gas)		Code Size Increase (B)		Additional Cost RW (US$)	
	RW	SM	RW	SM	per TX	per Upgrade
[BEC]	83	164	117 (1.0%)	133 (1.1%)	< 0.01	0.01
[SMT]	47	108	191 (0.8%)	97 (0.4%)	< 0.01	0.01
[UET]	225	21	1,299 (18.2%)	541 (7.6%)	< 0.01	0.071
[SCA]	47	0	3,811 (17.3%)	361 (1.6%)	< 0.01	0.189
[HXG]	120	541	997 (28.1%)	519 (14.6%)	< 0.01	0.057

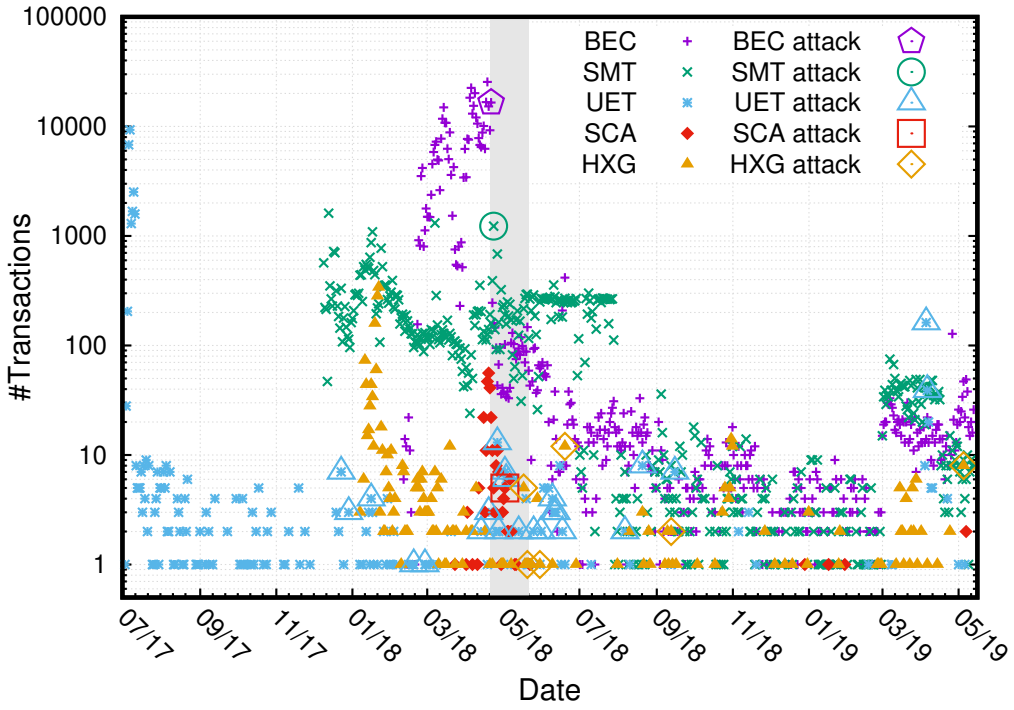


Figure 7.9: Activity timeline of each contract. The grey shadow indicates the time window in which the vulnerabilities of these contracts are disclosed by Peckshield [Pecc], and the big hollow points signify the occurrences of the attacks.

Detecting Attacks

The patch tester of EVMPatch allows us to identify any prior attack transactions. Figure 7.9 shows the timeline of the transactions we replayed and the attacks we identified using the patcher tester. We observe that while the vulnerabilities of the other token contracts have been reported within a fairly reasonable time after the first attack, the token UET was exploited five months before the public bug disclosure. More surprisingly, all contracts are still fairly active though they encountered a decrease in transaction volume after public disclosure of the vulnerabilities. Despite the fact that all of these vulnerabilities have been discovered around one year before the time of writing, there are still 23 630 transactions (4.66 % of the evaluated transactions) issued to these vulnerable contracts after the public disclosure of the vulnerabilities. This includes further attacks that have been successfully executed even after public disclosure. This means that the owners of those contracts did not properly migrate to patched versions, and users were not adequately notified of the vulnerable state of these contracts.

Analysis of False Alarms and Missed Bugs

During our analysis of the vulnerable contracts, we identified false alarms and missed bugs caused by the vulnerability reporting of Osiris [FSS18]. This demonstrates that our patch testing is an important step in the process, as many analysis tools are imprecise. We found that in the default configuration, Osiris often achieves limited code coverage. To this end, we utilized different timeout settings for both the whole analysis and for queries to the SMT solver [MB08] and combined the results of multiple runs to achieve better code coverage. Furthermore, we found that—contrary to the claims in the original *Osiris* paper [FSS18]—not all vulnerabilities are accurately detected by Osiris in two particular cases.

Hexagon (HXG) Token This contract is vulnerable to an integer overflow, which allows an attacker to transfer very large amounts of ERC-20 tokens [Pecb]. Osiris reports two false positives, which are caused by EVM code that is generated by the Solidity compiler. Even though all types are unsigned types in the Solidity source code, the compiler generates a signed addition. Here, Osiris reports a possible integer overflow when -2 is added to the *balanceOf* mapping variable. When performing signed integer additions with negative values, the addition naturally overflows when the result moves from the negative value range into the positive value range and vice versa. As such, EVMPatch patches a checked addition for an unsigned arithmetic operation which will always overflow. With our patch tester, we observe all the failing transactions and perform manual analysis of the patched contract's bytecode to determine that the root cause is an issue in the Solidity compiler, i.e., the generated code requires an additional instruction when compared to a simple unsigned subtraction.

The Hexagon Token contract is vulnerable to an integer overflow, which allows an attacker to transfer very large amounts of ERC-20 Tokens [Pecb]. Osiris reports two false positives, which are caused by EVM code that is generated by the Solidity compiler. Figure 7.10 shows the Solidity and corresponding EVM code. In the Solidity code in the upper listing of Figure 7.10, we can see that the variable `_value` is of type unsigned (Line 6) and the variable `burnPerTransaction` is also unsigned (Line 1). Even though all types are unsigned types in the addition in Line 9, the compiler generates a signed addition. The signed addition can be seen in lines 7 to 9 in the lower listing, where a negative signed value is pushed onto the stack. Here, Osiris reports a possible integer overflow when -2 is added to the *balanceOf* mapping variable. When performing signed integer additions with negative values, the addition naturally overflows when the result moves from the negative value range into the positive value range and vice versa.

When patching the `ADD` on line 12 in the lower listing of Figure 7.10, with a checked addition, we introduce a false positive. Replacing a signed addition with a checked unsigned addition will always fail if negative numbers are involved since they naturally trigger an overflow when switching between the positive and negative ranges due to the two's complement representation. The patch tester in our pipeline marked almost all transactions as failing, which is a strong indicator of a failed patch.

```
uint8 public constant burnPerTransaction = 2;
mapping (address => uint256) public balanceOf;
// ...
function _transfer(address _from,
                   address _to,
                   uint _value) internal {
// ...
// Line 85 in the Hexagon contract
balanceOf[_from] -= _value + burnPerTransaction;
// ...
```

```
// ...
DUP1
SLOAD // load balanceOf[_from]
PUSH1 0x1
NOT   // ~1 == 0xffffffff...fffffffffe == -2
SWAP1 // balanceOf[_from] on top
DUP8  // the passed parameter `_value`
SWAP1 // stack = [ _balanceOf[_from], _value, -2,  ... ]
SUB   // x = _balanceOf[_from] - _value
DUP2  // stack = [-2, _balanceOf[_from] - _value, -2, ...]
ADD   // -2 + (_balanceOf[_from] - _value)
// ...
```

Figure 7.10: Problematic Solidity line in the Hexagon contract (top listing). Solidity generates the EVM code in the bottom listing. Instead of subtracting 2 from an unsigned integer, Solidity promotes this to a signed integer and adds −2.

Social Chain (SCA) Our results also show a problem with Osiris when analyzing the SCA token. While Osiris does detect a possible overflow during multiplication in the problematic Solidity source code line, it does not detect the possible integer overflow for an addition in the same source code line. However, in the actual attack transaction, the integer overflow happens during the not-flagged addition operation. As such, this constitutes a false negative problem of Osiris. Since the vulnerable addition is not reported by Osiris, it is also not automatically patched by EVMPatch. In contrast, for the manually patched version we took both arithmetic operations into account. The related attack transaction was previously reported as an attack transaction [Pec18].

Summary of Evaluation To summarize, our evaluation on integer overflow detection shows that EVMPatch can correctly apply patches to smart contracts preventing *any* integer overflow attack. Furthermore, EVMPatch incurs only a *negligible* gas overhead during deployment and runtime, especially compared to the Ether at stake. Our analysis shows that the analyzed vulnerable smart contracts are *still in active use, even after being attacked* and the vulnerabilities being publicly disclosed. This motivates the need for a timely patching framework such as EVMPatch. Lastly, based on an extensive and detailed analysis of 26 385 532 transactions, we demonstrate

that EVMPatch always preserves the contract's original functionality except for a few cases, where the vulnerability report (generated by the third-party tool Osiris) was not accurate or bad coding practices were used (blackhole address).

7.5 Developer Study

In this Section we show the results of a developer study we conducted to gain insights into the usability of the EVMPatch framework. Using the developer study we assess the difficulty of manually patching and upgrading smart contracts. With our study we confirm that existing patching and deployment processes are too error-prone. For example, none of the developers correctly converted a contract into an upgradable contract. Furthermore, we show that EVMPatch can be easily used by developers to semi-manually patch their smart contracts, for example in case an access control bug is discovered.

Developer Background We conducted a study with 6 professional developers with varying prior experience in using blockchain technologies and developing smart contracts. Our developers consider themselves familiar with blockchain technologies but not very familiar with developing Solidity code. None of the developers have developed an upgradable contract before. As such, we can quantify the effort needed for a smart contract developer to learn and apply an upgradable contract pattern.

Methodology Throughout our study, we asked the developers to perform multiple tasks manually that are performed automatically by EVMPatch: (1) manually patch three contracts vulnerable due to integer overflow bugs given the output of a static analyzer (OSIRIS [FSS18]), (2) convert a contract to an upgradable contract manually and with EVMPatch, and (3) patch an access control bug using EVMPatch by writing a custom patch-template. The three tasks cover different scenarios, where EVMPatch can be useful to a developer. The first two tasks cover the use of EVMPatch to patch known bug classes with minimal human intervention. For these two tasks we assume no prior knowledge on patching smart contracts (see Table 7.5 how developers rated their prior experience with smart contracts). In contrast, the

Table 7.4: Timing results for the tasks as reported by the developers given in minutes and their reported confidence in the correctness of their results.

Task	Time (Minutes)			Confidence
	Median	Min	Max	Median (1-7)
Manual Integer Patches	47.50	35	78	6
Conversion	62.50	33	110	2.5
EVMPatch Conversion	1.50	1	3	-
Patch Template	4.00	2	15	7

third task consists of extending EVMPatch. This requires understanding a bug class and perform root cause analysis to properly patch the vulnerability. This is surely more challenging compared to the previous two tasks. Since the third task covers a different bug class, we believe there is no significant bias in the data due to the developers completing the other two tasks first.

For all tasks, we measured the time required by the developer to perform the task (excluding the time required for reading the tasks' description). We asked the developers to rate their familiarity with relevant technologies, their confidence levels in their patches, and the difficulty of performing the tasks on a 7-point Likert scale. The full questionnaire and the answers of the developers are shown in Table 7.5, and the recorded time measurements are shown in Table 7.4. We provide the supporting files in a github repository.[5]

We then performed both a manual code review and a cross-check with EVMPatch to analyze mistakes made by the developers. The results of our study show that significant effort is needed to correctly patch smart contracts manually, whereas EVMPatch enables simple, user-friendly, and efficient patching. The time measurements show that the developers, who had no prior experience with EVMPatch, were able to perform complex tasks utilizing EVMPatch within minutes.

Patching Integer Overflow Bugs We asked the developers to fix all integer overflow vulnerabilities in three contracts: 1 [BEC] (CVE-2018-10299, 299 lines of code), and 2 [HXG] (CVE-2018-11239, 102 lines of code), and 3 [SCA] (CVE-2018-10706, 404 lines of code). To provide a representative set of contracts, we chose three ERC-20 contracts with varying complexity (in terms of lines of code) and where the static analysis also includes missed bugs and false alarms (see Section 7.4). We ran OSIRIS on all three contracts and provided the developers the analysis output as well as a copy of the SafeMath Solidity library. This accurately resembles a real-world scenario, where a blockchain developer quickly needs to patch a smart contract based on the analysis results of recent state-of-the-art vulnerability analysis tools and can look-up manual patching tutorials available online. All developers manually and correctly patched the source code of all three contracts which demonstrates their expertise in blockchain development. However, on the downside, it took the developers on average 51.8 min ($\sigma = 16.6$ min) to create patched version for the three contracts. In contrast, EVMPatch fully automates the patching process and is able to generate patches for the three contracts within a maximum of 10 s.

Converting to an Upgradable Contract The developers had to convert a given smart contract into an upgradable smart contract. We provided the developers a short description of the delegatecall-proxy pattern and asked them to convert the given contract into two contracts: one proxy contract and a logic contract, which is based on the original contract. We provided no further information on how to handle the storage-layout problem, and we explicitly allowed using code found online. The developers required an average of 66.3 min to convert a contract into an upgradable contract, with $\sigma = 31.3$ min, the fastest taking 33 min, and the slowest developer taking

[5] github.com/uni-due-syssec/evmpatch-developer-study

110 min. *None* of the developers performed a correct conversion into an upgradable contract, which is also reflected in a median confidence of 2.5 in the correctness reported by the developers. We observed two major mistakes: (a) The proxy contract would only support a fixed set of functions, i.e., the proxy would not support adding functions to the contract, and (b) more importantly, only one out of six developers correctly handled storage collisions in the proxy and logic contract, i.e., five of the six converted contracts were broken by design. Hence, it remains open how long it would take developers to perform a correct conversion.

Next, we asked the developers to utilize EVMPatch to create and deploy an upgradable contract. As EVMPatch does not require *any* prior knowledge about upgradable contracts, the developers were able to deploy a correct upgradable contract within at most 3 min. In addition, patching with EVMPatch inspires high confidence—a median of 7, the best rating on our scale—in the correctness of the patch. This gives a strong confirmation that deployment of a proxy with EVMPatch is indeed superior to manual patching and upgrading.

Extending EVMPatch The developers had to write a custom patch template for EVMPatch. We instructed the developers on how to use EVMPatch and how patch templates are written with EVMPatch's patch template language (see Figure 7.7 for an example). Furthermore, we presented the developers an extended bug report that shows how an access control bug can be exploited. The developers leveraged the full EVMPatch system, i.e., EVMPatch applies the patch and validates the patch using the patch tester component which replays past transactions from the blockchain and notifies the developer whether: (a) the patch prevents a known attack, and (b) whether the patch broke functionality in other prior legitimate transactions. As such, EVMPatch allowed the developers to create a fully functional and securely patched upgradable contract within a few minutes. On average, the developers only needed 5.5 min, and a maximum of 15 min, to create a custom patch template. As expected, all developers correctly patched the given contract using EVMPatch, because a faulty patch would have been reported by EVMPatch's patch tester to the developer. EVMPatch's integrated patch tester gives the developers a high confidence into their patch. On average, the developers reported a confidence level of 6.6 ($\sigma = 0.4$), where 7 is the most confident. Furthermore, *none* of the developers considered writing such a custom patch template as particularly difficult.

Summary Our study provides confirmation that EVMPatch offers a high degree of automation, efficiency, and usability thereby freeing developers from manual and error-prone tasks. In particular, none of the six developers were able to produce a correct upgradable contract mainly due to the difficulty of preserving the storage-layout. Our study also confirms that extending EVMPatch with custom patch templates is a feasible task, even for developers that are unaware of the inner workings of EVMPatch.

Table 7.5: Developer study questionnaire and answers by six developers (A-F).

	Question	Answers							Scale
		A	B	C	D	E	F	Median	
Q1	Did you write Solidity code in the last two weeks?	no	no	no	no	yes	no		(yes/no)
Q2	Have you previously worked on a production-grade Solidity-based Ethereum contract?	yes	no	no	no	no	no		(yes/no)
Q3	Have you previously worked on a production-grade smart contract on another Blockchain Platform?	no	no	yes	no	yes	yes		(yes/no)
Q4	How familiar are you with Blockchain technologies in general?	6	5	7	6	6	6	**6**	(1 not familiar, 7 very familiar)
Q5	How familiar are you with the Ethereum Blockchain in particular?	6	5	4	2	6	2	**4.5**	(1 not familiar, 7 very familiar)
Q6	How familiar are you with the Solidity programming language?	6	3	2	1	5	1	**2.5**	(1 not familiar, 7 very familiar)
Q7	How familiar are you with upgradable contracts in Solidity?	5	3	1	1	4	1	**2**	(1 not familiar, 7 very familiar)
	Task 1								
T1Q1	How confident are you in the correctness of your patch to contract 1?	5	7	7	6	7	6	**6.5**	(1 least confident, 7 most confident)
T1Q2	How confident are you in the correctness of your patch to contract 2?	6	7	7	4	7	6	**6.5**	(1 least confident, 7 most confident)
T1Q3	How confident are you in the correctness of your patch to contract 3?	3	5	6	5	2	4	**4.5**	(1 least confident, 7 most confident)
T1Q4	How much time did you need to patch all three contracts?	78	35	40	40	55	63	**47.5**	(Time in Minutes)
	Task 2								
T2Q1	Have you previously used the delegatecall-proxy pattern in a Solidity contract?	no	no	no	no	no	no		(yes/no)
T2Q2	Have you previously used a different pattern to make a Solidity contract upgradable?	no	no	no	no	no	no		(yes/no)
T2Q3	Have you previously used a different upgradable smart contract?	no	no	no	no	no	no		(yes/no)
T2Q4	How confident are you in the correctness of your conversion?	5	3	1	1	5	2	**2.5**	(1 least confident, 7 most confident)
T2Q5	How difficult was the manual conversion?	4	5	5	6	4	6	**5**	(1 easy, 7 most difficult)
T2Q6	How difficult was the conversion using the evmpatch tool?	1	1	1	1	1	1	**1**	(1 easy, 7 most difficult)
T2Q8	How much time did you need to convert the contract to an upgradable contract (Step 1)?	110	80	45	90	40	33	**62.5**	(Time in Minutes)
T2Q8	How much time did you need to convert the contract using EVMPatch (Step 2)?	3	1	1	2	3	1	**1.5**	(Time in Minutes)
	Task 3								
T3Q1	How confident are you in the correctness of your patch?	6	7	7	7	7	6	**7**	(1 least confident, 7 most confident)
T3Q2	How difficult was the conversion using the EVMPatch tool?	2	1	1	1	1	1	**1**	(1 easy, 7 most difficult)
T3Q3	How much time did you need to create and deploy the patch using EVMPatch?	15	2	5	2	6	3	**4**	(Time in Minutes)

7.6 Related Work

Bytecode rewriting for patching smart contracts has also been explored by Zhang et al. [Zha+20]. SMARTSHIELD requires a complete CFG to update jump targets and data references. As discussed in Section 7.1.2, generating a highly accurate CFG is highly challenging due to the EVM's bytecode format. We believe that such a bytecode rewriting strategy does not scale to larger and more complicated contracts. In contrast, EVMPatch's trampoline-based rewriting strategy does not require an accurate CFG and is much more resilient when rewriting complex contracts. Furthermore, SMARTSHIELD implements custom bytecode analysis to detect vulnerabilities, which may not be as accurate as specialized analyses. For example, SMARTSHIELD's analysis does not infer whether an integer type is signed, which is important for accurate integer overflow detection [FSS18]. In contrast, EVMPatch is a flexible framework that can integrate many static analysis tools for detecting vulnerabilities and can leverage analysis tool improvements with minimal effort. Last and most importantly, EVMPatch automates the whole lifecycle of deploying and managing an upgradable contract, while SMARTSHIELD is designed to harden a contract pre-deployment. With EVMPatch, a smart contract developer can also patch vulnerabilities that are discovered after deployment of the contract.

Torres et al. [TJS22] introduces *Elysium*, a context-aware patch generation framework for smart contracts that also utilizes bytecode rewriting. Similar to SMARTSHIELD, Elysium requires a complete CFG to introduce patches, and as such, is less resilient than EVMPatch's trampoline-based approach. However, in contrast to EVMPatch, Elysium features several static analysis passes to gather semantic context. For example, Elysium recovers integer signedness before patching integer-related bugs. In contrast, EVMPatch gathers this context from the underlying analysis tool *Osiris* [FSS18]. Furthermore, Elysium recovers the last used storage slot for smart contract global variables to store reentrancy lock variables. Similar to the source-level *ReentrancyGuard* patch, Elysium applies a single global reentrancy lock. EVMPatch is capable of creating the same type of patch but does not require contextual analysis. Instead, EVMPatch's patch template simply utilizes a randomly selected arbitrary address for lock variables. Essentially this ensures storage-layout compatibility in a probabilistic manner, relying on the vast number of storage slots. More specifically, the EVM storage area features 2^{256} different slots, and as such, it is highly unlikely to have colliding storage addresses when choosing them at random. Similarly, the Solidity compiler relies on the size of the storage address space to avoid collisions when implementing hash maps (i.e., the *mapping* type in Solidity). Finally, Elysium automatically inserts access control checks by introducing a special *owner* slot during contract construction and then augments functions with access control checks. However, it is not clear whether this approach to patching access control issues is always applicable. Elysium does not infer any semantic knowledge about existing access control mechanisms. As such, existing ownership transferral mechanisms would break due to the newly introduced access control checks. Furthermore, there are also more complex ownership mechanisms, such as so-called *multi-owned* contracts, which are owned by more than one owner and often require majority votes from owners before performing an action such as Ether transfer. Automatically introducing access control checks for a single

owner would break the semantics of the contract and potentially even introduce a vulnerability. To summarize, we conclude that Elysium does not properly utilize contextual information for patching. Especially for access control bugs, Elysium does not infer anything about existing access control mechanisms, potentially breaking smart contracts. In contrast, EVMPatch uses fully automatic patching for vulnerabilities like integer overflows and exposes an interface to allow manual specification of access control patches.

The Ethereum community explored several design patterns to allow upgradable smart contracts [Cona; Nad; Tra18; ZepOS] with manual migration to a new contract and the proxy pattern being the most popular (see Section 7.1.1). The ZeppelinOS [ZepOS] framework supports upgradable contracts by implementing the delegatecall-proxy pattern. However, developers have to manually ensure compatibility of the legacy and patched contract on the Solidity level. This can be achieved using static analysis tools that perform "upgradeability" checks (e.g., Slither [SliU] checks for a compatible storage layout), which relies on accurate knowledge of compiler behavior with respect to storage allocations. On the other hand, EVMPatch combines existing analysis tools and provides an automatic method to patch detected vulnerabilities while keeping the storage layout consistent by design. Moreover, using bytecode rewriting prevents potential problems caused by unexpected behaviors from compilers.

7.7 Discussion and Conclusion

Updating erroneous smart contracts constitutes one of the major challenges in the field of blockchain technologies. The recent past has shown that attackers are fast in successfully abusing smart contract errors due to the natural design of the underlying technology: always online and available, one common and simple computing engine without any subtle software and configuration dependencies, and often a high amount of cryptocurrency at risk. While many proposals have introduced frameworks to aid developers in finding bugs [FSS18; KR18a; Luu+16; Sch+20b; Tsa+18], it remains open how developers and the community can quickly and automatically react to vulnerabilities on already deployed contracts. In this chapter, we describe the design and implementation of a framework that supports automated and nearly instant patching of smart contract errors based on bytecode rewriting. We implement a fully automated pipeline to identify and patch integer overflow bugs in smart contracts using our EVMPatch framework and the Osiris analysis tool. Our evaluation shows that EVMPatch is highly effective in patching bugs and also verifying patches. Beyond integer overflows, discuss potential patching mechanisms for reentrancy bugs and mishandled exceptions. We also show how to utilize EVMPatch for semi-manual patching of smart contracts for access control bugs(Section 7.3.4). We believe that already such a semi-manual approach is useful to developers, as they can quickly write new patch specifications and benefit from EVMPatch's patch testing infrastructure. With our developer study, we show that developers are able to write new patch specifications in mere minutes.

However, when dealing with access control bugs, EVMPatch's limitation becomes apparent: it relies on existing analysis tools to provide enough information to synthesize

a patch. Several bug classes, such as integer bugs or unhandled exceptions, are easy to patch as the patches require little semantic information about the contract. However, when it comes to synthesizing patches for access control, the patch template requires semantic knowledge about the contract, more specifically, how it implements access control. Unfortunately, there is no standard implementation of access control for Ethereum smart contracts and various different access control models are available. Furthermore, no current analysis tool is capable of providing semantic knowledge about the access control checks of smart contracts. As future work, existing analysis tools must be expanded to acquire semantic knowledge, which in turn can feed into EVMPatch to create patches for a more diverse set of smart contracts.

We were able to demonstrate that real-world integer overflow vulnerabilities can be successfully patched without violating the functional correctness of the smart contract. Our developer study shows that an automated patching approach greatly reduces the time required for patching smart contracts and that our implementation, EVMPatch, can be practically integrated into a smart contract developer's workflow. We believe that automated patching will increase the trustworthiness and acceptance of smart contracts as it allows developers to react quickly to reported vulnerabilities.

CHAPTER 8

CONCLUSION

This dissertation advances the state-of-the-art in the field of software security of two new secure execution environments: trusted enclaves and smart contracts. This dissertation shows how to adapt well-known techniques, such as symbolic execution, fuzzing, taint tracking, and binary rewriting, to new execution environments. The methods and tools presented in this dissertation allow software system vendors to significantly improve their products' software security by (1) identifying security vulnerabilities early in the development process and (2) hardening existing deployments.

Securely developing software for trusted execution environments, such as SGX enclaves, is highly challenging due to the new and complex threat model that such software faces. Current software analysis tools cannot identify security vulnerabilities that arise on the boundary of the enclave code and the untrusted part of the software system, as they only identify generic software security issues. They are unaware of the enclave-specific threat model and therefore miss crucial security vulnerabilities. To tackle this lack of analysis methods, Chapter 4 presents TeeRex, a methodology for automated vulnerability detection based on symbolic execution tailored specifically to trusted execution environments, such as SGX enclaves. We show how to adapt a general symbolic execution framework to the particularities of SGX, making TeeRex the first practical analysis tool that captures enclave-specific vulnerabilities. This chapter shows how to overcome challenges in implementing efficient and practical symbolic execution of SGX enclaves. Furthermore, we show how to exploit the particularities of the SGX technology to speed up symbolic execution with multiple CPU cores. Using this methodology, enclave developers can now automatically scan their SGX enclave for vulnerabilities before deploying them.

To evaluate our symbolic execution tool TeeRex, we perform an analysis of several publicly available enclaves. Out of 6 enclaves, we identify critical vulnerabilities in a set of 5 enclaves. This includes widely deployed enclaves such as fingerprint reader drivers used on Lenovo and Dell laptops. This demonstrates the practical usefulness of TeeRex as an analysis tool. Furthermore, it shows that current enclaves often disregard the complex threat model of the SGX technology, leading to vulnerabilities.

Based on the findings in these enclaves, we perform root-cause analysis of the bugs that TeeRex identified. We systematize these root causes and develop a set

of vulnerability patterns. For the first time, offering enclave developers a systematic analysis of root causes and guidelines to avoid code anti-patterns in the first place. To validate the security of existing enclaves, developers can utilize the analysis techniques pioneered in the TeeRex tool.

With TeeRex, developers for enclaves already have a tool at hand to improve the security of SGX enclaves by systematically vetting the interface of the enclave for SGX specific security vulnerabilities. However, TeeRex is based on symbolic execution and often fails to uncover vulnerabilities deep inside large code bases due to the state explosion problem. As such, other analysis directions need to be explored in the future. For example, coverage-guided fuzzing or hybrids of fuzzing and concolic execution are prime candidates to adapt for analyzing trusted enclave code.

On the other hand, SGX is only one of many popular TEE technologies. In fact, SGX is being deprecated in favor of other technologies, such as Intel's Trusted Domain Extensions (TDX), AMD's Secure Encrypted Virtual machines (SEV), or ARM's Realm Management Extensions (RME). These extensions all provide the possibility of launching full virtual machines inside enclaves. While this increases the familiarity of developers with the TEE, it increases the size of the TCB to a full operating system. It remains an interesting future research direction to assess whether these new TEEs are easier to secure than SGX enclave code.

Continuing the line of work on vulnerability detection, Chapter 5 shows how to improve the state-of-the-art in smart contract fuzzing. Chapter 5 presents an assessment of the capabilities of existing analysis tools. Here a special focus is placed on the capability to scale to long transaction sequences: an essential prerequisite to identify bugs in complex smart contracts. Based on our experiments, we conclude that existing smart contract analysis tools are not well equipped to handle the ever-increasing complexity of smart contract code.

To counteract this tendency and provide smart contract developers with a practical analysis tool, Chapter 5 presents EF↯CF, a high-performance smart contract fuzzer. With EF↯CF, we show how to apply state-of-the-art fuzzing techniques to the area of smart contract fuzzing. Using high-performance fuzzing techniques, EF↯CF is able to outperform existing fuzzers and, in practice, even the capabilities of symbolic analysis tools.

Based on the high-performance fuzzing framework, we show how to adapt a fuzzer to rigorously test Ethereum smart contracts in the face of the many-faceted and complex interactions that are possible in Ethereum. More specifically, we show how to accurately simulate malicious attacker-controlled smart contracts in a fuzzer. This allows EF↯CF to accurately detect reentrancy and compositional vulnerabilities in smart contracts. In contrast to all prior analysis tools, EF↯CF does not perform any heuristic detection of reentrancy vulnerabilities. Instead, EF↯CF utilizes a novel probabilistic method to simulate arbitrary attacker-controlled smart contracts that are part of a reentrancy attack against the target smart contract. This means that EF↯CF eradicates any false alarms caused by heuristic reentrancy detection and generates inspectable and replayable transaction sequences that exploit a reentrancy vulnerability. As such, EF↯CF is the first purely fuzzing-based exploit generator for smart contracts, facilitated by the groundwork on high-performance fuzzing of smart contracts presented at the beginning of Chapter 5.

The EF/CF prototype is designed as a high-performance fuzzer for EVM smart contracts. However, the general concepts and the architecture of EF/CF is independent of the targeted blockchain system. For example, EF/CF's design could be leveraged in an implementation of a high throughput fuzzer for other popular smart contract execution environments, such as the *rBPF* in the Solana blockchain or *WebAssembly* employed by Web Browsers and other Blockchain systems. EF/CF demonstrates that analysis tools must put significant effort into efficient input generation strategies. EF/CF outperforms even symbolic execution and hybrid concolic fuzzing tools. However, EF/CF essentially performs random search on the set of possible transaction sequences, guided only by code coverage feedback. As our experiments show, this still achieves very good results in practice because of EF/CF's optimized throughput. However, there are many techniques that attempt to prune the search space or perform more directed mutations, allowing a fuzzer to analyze targets even with slow throughput. It remains to be seen whether such techniques can be implemented efficiently, such that high throughput fuzzers such EF/CF can also utilize those techniques, and whether this actually improves the *time-to-bug* metric in practice.

However, all practical vulnerability detection approaches are prone to missing bugs. As such, the security of software cannot rely solely on fuzzing and symbolic execution to fix vulnerabilities before deployment. Developers require ways to deal with security issues after deployment. This is especially true for smart contracts, which are immutable by design, i.e., they are identified using a cryptographic hash of their code and initial state. Chapter 6 presents Sereum, a runtime attack detection approach tailored to reentrancy attacks on smart contracts. We show that Sereum is capable of accurately identifying reentrancy attacks at runtime. By replaying large parts of the transactions recorded on the Ethereum blockchain, we are able to show that Sereum correctly identifies several real-world reentrancy attacks. Furthermore, we uncovered new attacks that were not widely known in the community before our evaluation.

While Sereum is an effective tool to mitigate attacks across a blockchain system, it is highly challenging to integrate such a mitigation into an already deployed blockchain system, as it would change the execution semantics of the execution environment. As such, a more practical approach is required to react to newly discovered smart contract vulnerabilities or attacks. In Chapter 7, we present EVMPatch, a streamlined and automated approach to patching smart contracts using bytecode rewriting and the so-called proxy pattern to work around the immutability of smart contracts and facilitate upgrades. We show that EVMPatch can easily be used by developers to set up upgradable contracts and patch a diverse set of vulnerabilities, both based on manual patch specification and also using automated analysis tools. Both the vulnerability identification with EF/CF in Chapter 5 and the attack detection of Sereum in Chapter 6 can be utilized as input to EVMPatch's automated patching process. Thus, EVMPatch allows smart contract developers to bring the power of existing analysis tools to already deployed contracts.

Overall, we conclude that *secure execution environments* give certain security guarantees to developers. However, these execution environments also pose new and unfamiliar threat models, making them more likely to introduce security vulnerabilities. Given the abundance of vulnerabilities, which we and the community at

large discovered, it is clear that there is a lack of awareness of these security issues and, arguably even worse, existing tooling is not adequate to assist developers in their efforts to secure critical code in enclaves and smart contracts. With TeeRex and EF↯CF, we show how to adapt vulnerability identification techniques to identify security vulnerabilities specific to secure execution environments before deployment. With Sereum and EVMPatch, we provide methods to securely operate software in these new secure execution environments, overcoming the limitations of the execution environments to facilitate mitigation of vulnerabilities.

BIBLIOGRAPHY

List of Publications

Complete list of publications with contributions of the author of this dissertation over the course of his PhD studies.

[Ade+20] Sridhar Adepu, Ferdinand Brasser, Luis Garcia, Michael Rodler, Lucas Davi, Ahmad-Reza Sadeghi, and Saman A. Zonouz. "Control Behavior Integrity for Distributed Cyber-Physical Systems". In: *11. ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2020. DOI: 10.1109/ICCPS48487.2020.00011.

[Ber+22] Lukas Bernhard, Michael Rodler, Thorsten Holz, and Lucas Davi. "xTag: Mitigating Use-After-Free Vulnerabilities via Software-Based Pointer Tagging on Intel x86-64". In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2022. DOI: 10.1109/EuroSP53844.2022.00038.

[CRD20] Tobias Cloosters, Michael Rodler, and Lucas Davi. "TeeRex: Discovery and Exploitation of Memory Corruption Vulnerabilities in SGX Enclaves". In: *29th USENIX Security Symposium*. USENIX Association, 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/cloosters.

[Dra+23] Oussama Draissi, Tobias Cloosters, David Klein, Marius Musch, Michael Rodler, Lucas Davi, and Martin Johns. "Dissecting and Fuzzing Native Code on the Web". In: (2023). Under submission.

[Gie+22] Jens-Rene Giesen, Sebastian Andreina, Michael Rodler, Ghassan O. Karame, and Lucas Davi. "Practical Mitigation of Smart Contract Bugs". In: *CoRR* abs/2203.00364 (2022). DOI: 10.48550/arXiv.2203.00364. arXiv: 2203.00364. Under submission.

[Paa+21a] David Paaßen, Sebastian Surminski, Michael Rodler, and Lucas Davi. "My Fuzzer Beats Them All! Developing a Framework for Fair Evaluation and Comparison of Fuzzers". In: *26th European Symposium on Research in Computer Security (ESORICS)*. Lecture Notes in Computer Science. Springer, 2021. DOI: 10.1007/978-3-030-88418-5_9.

[Rod+19] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. "Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks". In: *26th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2019. URL: https://www.ndss-symposium.org/ndss-paper/sereum-protecting-existing-smart-contracts-against-re-entrancy-attacks/.

[Rod+21] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. "EVMPatch: Timely and Automated Patching of Ethereum Smart Contracts". In: *30th USENIX Security Symposium*. USENIX Association, 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/rodler.

[Rod+23] Michael Rodler, David Paaßen, Wenting Li, Lukas Bernhard, Thorsten Holz, Ghassan O. Karame, and Lucas Davi. "EF/CF: A High Performance Fuzzer for Ethereum Smart Contracts". In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2023. DOI: 10.1109/EuroSP57164.2023.00034.

References

[MKRh] URL: https://github.com/nexusdev/hack-recovery (visited on 07/28/2018).

[Aba+05] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow Integrity". In: *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*. 2005. DOI: 10.1145/1102120.1102165.

[Aba+09] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow integrity principles, implementations, and applications". In: *ACM Trans. Inf. Syst. Secur.* 13.1 (2009). DOI: 10.1145/1609956.1609960.

[Adv+91] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. "Detecting Data Races on Weak Memory Systems". In: *Proceedings of the 18th Annual International Symposium on Computer Architecture*. ACM, 1991, pp. 234–243. DOI: 10.1145/115952.115976.

[Ait02] Dave Aitel. *The advantages of block-based protocol analysis for security testing*. Tech. rep. 2002. URL: http://www.immunityinc.com/downloads/advantages_of_block_based_analysis.pdf.

[Alb+18] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. "EthIR: A Framework for High-Level Analysis of Ethereum Bytecode". In: Lecture Notes in Computer Science 11138 (2018), pp. 513–520. DOI: 10.1007/978-3-030-01090-4_30.

[Alb+20] Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. "Taming callbacks for smart contract modularity". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020). DOI: 10.1145/3428277.

[Ald+20] Fritz Alder, Jo Van Bulck, David F. Oswald, and Frank Piessens. "Faulty Point Unit: ABI Poisoning Attacks on Intel SGX". In: *ACSAC '20: Annual Computer Security Applications Conference*. ACM, 2020. DOI: 10.1145/3427228.3427270.

[And20] Ross J. Anderson. *Security engineering - a guide to building dependable distributed systems (3. ed.)* Wiley, 2020. ISBN: 978-111-964-2-7-8-7.

[AB14] Andrea Arcuri and Lionel Briand. "A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering". In: *Software Testing, Verification and Reliability* (2014).

[Asc+19a] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. "NAUTILUS: Fishing for Deep Bugs with Grammars". In: *NDSS*. 2019. DOI: 10.14722/ndss.2019.23. URL: http://dx.doi.org/10.14722/ndss.2019.23.

[Asc+19b] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. "REDQUEEN: Fuzzing with Input-to-State Correspondence". In: *Proceedings 2019 Network and Distributed System Security Symposium*. NDSS. Internet Society, 2019. DOI: 10.14722/ndss.2019.23371.

[Asy] Asylo Authors. *Asylo - An open and flexible framework for enclave applications*. URL: https://asylo.dev/ (visited on 02/01/2022).

[ABC17] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. "A Survey of Attacks on Ethereum Smart Contracts (SoK)". In: *International Conference on Principles of Security and Trust*. Springer. 2017.

[Aub+17] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. *TaLoS: Secure and Transparent TLS Termination inside SGX Enclaves*. en. Tech. rep. 2017/5. Imperial College London, Mar. 2017. URL: https://www.doc.ic.ac.uk/research/technicalreports/2017/DTRS17-5.pdf.

[Avg+14] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. "Enhancing symbolic execution with veritesting". In: *36th International Conference on Software Engineering*. ACM, 2014. DOI: 10.1145/2568225.2568293.

[Bab+19] Domagoj Babic, Stefan Bucur, Yaohui Chen, Franjo Ivancic, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. "FUDGE: fuzz driver generation at scale". In: *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/SIGSOFT FSE. ACM, 2019. DOI: 10.1145/3338906.3340456.

[Bal+18] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. "A Survey of Symbolic Execution Techniques". In: *ACM Comput. Surv.* 51.3 (May 2018). ISSN: 0360-0300. DOI: 10.1145/3182657.

[Bar+11] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4". In: *Computer Aided Verification - 23rd International Conference (CAV)*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14.

[BT18] Clark W. Barrett and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Springer, 2018, pp. 305–343. DOI: 10.1007/978-3-319-10575-8_11.

[Bas+17] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. "Synthesizing program input grammars". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI. ACM, 2017. DOI: 10.1145/3062341.3062349.

[BPH14] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. "Shielding Applications from an Untrusted Cloud with Haven". In: *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI*. 2014. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann.

[BEC] *BeautyChainToken*. URL: https://etherscan.io/address/0xc5d105e63711398af9bbff092d4b6769c82f793d.

[Bek+12] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. "A Taint Based Approach for Smart Fuzzing". In: *Fifth IEEE International Conference on Software Testing, Verification and Validation*. ICST. IEEE Computer Society, 2012. DOI: 10.1109/ICST.2012.182.

[Bio+18] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. "The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX". In: *27th USENIX Security Symposium, USENIX Security*. 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/biondo.

[Bla+19] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. "GRIMOIRE: Synthesizing Structure while Fuzzing". In: *28th USENIX Security Symposium*. USENIX Association, 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/blazytko.

[REVST] BlockSec. *Revest Finance Vulnerabilities: More than Re-entrancy*. Mar. 2022. URL: https://blocksecteam.medium.com/revest-finance-vulnerabilities-more-than-re-entrancy-1609957b742f (visited on 06/07/2022).

[BD95] Hans-J. Boehm and Paul Dubois. "Dynamic memory allocation and garbage collection". In: *Computers in Physics* 9.3 (1995), pp. 297–303.

[BCD21a] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. "Fuzzing Symbolic Expressions". In: *43rd IEEE/ACM International Conference on Software Engineering*. ICSE. IEEE, 2021. DOI: 10.1109/ICSE43902.2021.00071.

[BCD21b] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. "FUZZOLIC: Mixing fuzzing and concolic execution". In: *Comput. Secur.* 108 (2021), p. 102368. DOI: 10.1016/j.cose.2021.102368.

[Bos+22] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. "SAILFISH: Vetting Smart Contract State-Inconsistency Bugs in Seconds". In: *43rd IEEE Symposium on Security and Privacy*. S&P. IEEE, 2022. DOI: 10.1109/SP46214.2022.9833721.

[BEL75] Robert S Boyer, Bernard Elspas, and Karl N Levitt. "SELECT—a formal system for testing and debugging programs by symbolic execution". In: *ACM SigPlan Notices* 10.6 (1975). URL: https://dl.acm.org/citation.cfm?id=808445.

[Bra+17] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. "Software Grand Exposure: SGX Cache Attacks Are Practical". In: *USENIX Workshop on Offensive Technologies*. 2017.

[Bre+17] Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. *An In-Depth Look at the Parity Multisig Bug*. 2017. URL: http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/ (visited on 04/20/2018).

[Bre+18] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. "Vandal: A Scalable Security Analysis Framework for Smart Contracts". In: *arXiv:1809.03981* (2018). arXiv: 1809.03981 [cs.PL]. URL: http://arxiv.org/abs/1809.03981.

[Bro22] Martin Holst Swende Brooklyn Zelenka, Greg Colvin. *EIP 2315 - Simple Subroutines for the EVM*. 2022. URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2315.md (visited on 01/14/2022).

[BH00] Bryan Buck and Jeffrey K Hollingsworth. "An API for Runtime Code Patching". In: *Int. J. High Perform. Comput. Appl.* 14.4 (Nov. 2000). ISSN: 1094-3420. DOI: 10.1177/109434200001400404.

[Bur67] W H Burkhardt. "Generating test programs from syntax". In: (Mar. 1967). DOI: 10.1007/BF02235512.

[But16] Vitalik Buterin. *EIP-170: Contract code size limit*. 2016. URL: https://eips.ethereum.org/EIPS/eip-170 (visited on 02/17/2022).

[CREAM] C.R.E.A.M Finance. *C.R.E.A.M. Finance Post Mortem: AMP Exploit*. Sept. 2021. URL: https://medium.com/cream-finance/c-r-e-a-m-finance-post-mortem-amp-exploit-6ceb20a630c5 (visited on 06/07/2022).

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI*. 2008. URL: http://www.usenix.org/events/osdi08/tech/full%5C_papers/cadar/cadar.pdf.

[Cad+08] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. "EXE: Automatically Generating Inputs of Death". In: *ACM Trans. Inf. Syst. Secur.* 12.2 (2008). DOI: 10.1145/1455518.1455522.

[Can+19] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "A Systematic Evaluation of Transient Execution Attacks and Defenses". In: *28th USENIX Security Symposium*. USENIX Association, 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/canella.

[Cec+21] Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew C. Myers. "Compositional Security for Reentrant Applications". In: *42nd IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2021. DOI: 10.1109/SP40001.2021.00084.

[Cha+12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. "Unleashing Mayhem on Binary Code". In: *2012 IEEE Symposium on Security and Privacy*. IEEE, May 2012. DOI: 10.1109/SP.2012.31.

[CS13] Stephen Checkoway and Hovav Shacham. "Iago attacks: why the system call API is a bad untrusted RPC interface". In: *ASPLOS*. Vol. 13. 2013. DOI: 10.1145/2499368.2451145.

[Che+19a] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. "SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution". In: *IEEE European Symposium on Security and Privacy*. EuroS&P. IEEE, 2019. DOI: 10.1109/EuroSP.2019.00020.

[Che+22] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. "JIGSAW: Efficient and Scalable Path Constraints Fuzzing". In: *2022 IEEE Symposium on Security and Privacy*. 2022. DOI: 10.1109/SP46214.2022.00102.

[CC18] Peng Chen and Hao Chen. "Angora: Efficient Fuzzing by Principled Search". In: *2018 IEEE Symposium on Security and Privacy*. S&P. IEEE Computer Society, 2018. DOI: 10.1109/SP.2018.00046.

[Che+21] Ting Chen, Zihao Li, Xiapu Luo, Xiaofeng Wang, Ting Wang, Zheyuan He, Kezhao Fang, Yufei Zhang, Hang Zhu, Hongwei Li, Yan Cheng, and Xiao-Song Zhang. "SigRec: Automatic Recovery of Function Signatures in Smart Contracts". In: *IEEE Trans. Software Eng.* (2021). ISSN: 1939-3520. DOI: 10.1109/TSE.2021.3078342.

[Che+19b] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. "EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers". In: *28th USENIX Security Symposium*. 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/chen-yuanliang.

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E: a platform for in-vivo multi-path analysis of software systems". In: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*. Ed. by Rajiv Gupta and Todd C. Mowry. ACM, 2011, pp. 265–278. DOI: 10.1145/1950365.1950396.

[Cho+19] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. "Grey-box concolic testing on binary code". In: *Proceedings of the 41st International Conference on Software Engineering*. ICSE. IEEE / ACM, 2019. DOI: 10.1109/ICSE.2019.00082.

[Cho+21] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. "SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses". In: *36th IEEE/ACM International Conference on Automated Software Engineering*. ASE. IEEE, 2021. DOI: 10.1109/ASE51524.2021.9678888.

[CH00] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP. 2000. DOI: 10.1145/351240.351266.

[CLO07a] James Clause, Wanchun Li, and Alessandro Orso. "Dytan: A Generic Dynamic Taint Analysis Framework". In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. ACM, 2007. DOI: 10.1145/1273463.1273490.

[CLO07b] James A. Clause, Wanchun Li, and Alessandro Orso. "Dytan: a generic dynamic taint analysis framework". In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*. 2007. DOI: 10.1145/1273463.1273490.

[Clo+22] Tobias Cloosters, Johannes Willbold, Thorsten Holz, and Lucas Davi. "SGXFuzz: Efficiently Synthesizing Nested Structures for SGX Enclave Fuzzing". In: *31st USENIX Security Symposium*. USENIX Association, Aug. 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/cloosters.

[Cona] ConsenSys. *Ethereum Smart Contract Best Practices: Upgradeability*. URL: https://consensys.github.io/smart-contract-best-practices/development-recommendations/precautions/upgradeability/ (visited on 02/17/2022).

[Conb] ConsenSys Diligence. *Ethereum Smart Contract Best Practices*. URL: https://consensys.github.io/smart-contract-best-practices/known_attacks/ (visited on 07/25/2018).

[Con+21] Filippo Contro, Marco Crosara, Mariano Ceccato, and Mila Dalla Preda. "EtherSolve: Computing an Accurate Control-Flow Graph from Ethereum Bytecode". In: *29th IEEE/ACM International Conference on Program Comprehension*. ICPC. IEEE, 2021. DOI: 10.1109/ICPC52881.2021.00021.

[CD16] Victor Costan and Srinivas Devadas. "Intel SGX Explained". In: *IACR Cryptology ePrint Archive* (2016). URL: http://eprint.iacr.org/2016/086.

[Cow+98] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks". In: *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998*. USENIX Association, 1998. URL: https://www.usenix.org/conference/7th-usenix-security-symposium/stackguard-automatic-adaptive-detection-and-prevention.

[MKRr] *Critical ether token wrapper vulnerability - ETH tokens salvaged from potential attacks*. June 2016. URL: https://www.reddit.com/r/MakerDAO/comments/4niu10/critical_ether_token_wrapper_vulnerability_eth/ (visited on 07/28/2018).

[Cry] Crytic / Trail of Bits. *ethersplay: EVM disassembler and related analysis tools*. URL: https://github.com/crytic/ethersplay/.

[Cyb] CyberLink. *PowerDVD Ultra Requirements*. URL: https://www.cyberlink.com/products/powerdvd-ultra/spec_en_US.html (visited on 11/14/2019).

[Dai16a] Phil Daian. *Analysis of the DAO exploit*. 2016. URL: http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

[Dai16b] Philip Daian. *Chasing the DAO Attacker's Wake*. June 2016. URL: https://pdaian.com/blog/chasing-the-dao-attackers-wake/ (visited on 07/26/2017).

[Dav+12] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. "MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones". In: *Proceedings Network and Distributed System Security Symposium (NDSS)*. 2012.

[Dev22] The Rust Fuzz Project Developers. *Arbitrary - The trait for generating structured data from arbitrary, unstructured input.* Version v1.1.3. 2022. URL: https://github.com/rust-fuzz/arbitrary/ (visited on 06/03/2022).

[Dua+] Ran Duan, Long Li, Shi Jia, Yu Ding, Yulong Zhang, Yueqiang Cheng, Lenx Wei, and Tanghui Chen. *Apache Teaclave Rust-SGX SDK - Samplecode "tls/tlsclient"*. URL: https://github.com/apache/incubator-teaclave-sgx-sdk/tree/master/samplecode/tls/tlsclient (visited on 02/28/2020).

[Dur02] Tyler Durden. "Bypassing PaX ASLR protection". In: *Phrack Magazine* 59.9 (2002). URL: http://phrack.org/issues/59/9.html.

[Dur+20] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. "Empirical review of automated analysis tools on 47,587 Ethereum smart contracts". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE. 2020. DOI: 10.1145/3377811.3380364.

[Dut14] Bruno Dutertre. "Yices 2.2". In: *Computer Aided Verification - 26th International Conference (CAV)*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 737–744. DOI: 10.1007/978-3-319-08867-9_49.

[Eck+18] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. "HeapHopper: Bringing Bounded Model Checking to Heap Implementation Security". In: *27th USENIX Security Symposium*. USENIX Association, 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/eckert.

[eEVM] Microsoft. *Enclave EVM*. URL: https://github.com/microsoft/eEVM.

[TaLoS] *Efficient TLS termination inside Intel SGX enclaves for existing applications: lsds/TaLoS*. Aug. 7, 2019. URL: https://github.com/lsds/TaLoS (visited on 08/27/2019).

[EIP] Ethereum EIPs. *ERC930 - Eternal Storage Standard*. https://github.com/ethereum/EIPs/issues/930. [Online; accessed 2019-11-08].

[Enc+14] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones". In: *ACM Transactions on Computer Systems (TOCS)* 32.2 (2014). DOI: 10.1145/2619091.

[Eth22] Etherscan. *Top Accounts by ETH Balance*. 2022. URL: https://etherscan.io/accounts (visited on 01/14/2022).

[HODLW] *etherscan.io: HODLWallet*. URL: https://etherscan.io/address/0x4a8d3a662e0fd6a8bd39ed0f91e4c1b729c81a38 (visited on 12/12/2021).

[INSD] *etherscan.io: InstaDice*. URL: https://etherscan.io/address/0xfe1b613f17f984e27239b0b2dccfb1778888dfae (visited on 12/12/2021).

[SpCLC] *etherscan.io: LedgerChannel (SpankChain)*. URL: https://etherscan.io/address/0xf91546835f756da0c10cfa0cda95b15577b84aa7 (visited on 12/12/2021).

[SYES] *etherscan.io: SysEscrow*. URL: https://etherscan.io/address/0x903643251af408a3c5269c836b9a2a4a1f04d1cf (visited on 12/12/2021).

[DAO] *etherscan.io: The DAO contract*. URL: https://etherscan.io/address/0xBB9bc244D798123fDe783fCc1C72d3Bb8C189413 (visited on 08/01/2018).

[imBTC] *etherscan.io: The Tokenized Bitcoin (imBTC)*. URL: https://etherscan.io/address/0x3212b29E33587A00FB1C83346f5dBFA69A458923 (visited on 12/12/2021).

[FGG19] Josselin Feist, Gustavo Grieco, and Alex Groce. "Slither: a static analysis framework for smart contracts". In: *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*. WETSEB@ICSE. IEEE / ACM, 2019. DOI: 10.1109/WETSEB.2019.00008.

[FSS18] Christof Ferreira-Torres, Julian Schütte, and Radu State. "Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts". In: *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*. 2018. DOI: 10.1145/3274694.3274737.

[Fio+20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. "AFL++: Combining incremental steps of fuzzing research". In: *14th USENIX Workshop on Offensive Technologies*. WOOT. 2020. URL: https://aflplus.plus//papers/aflpp-woot2020.pdf.

[Fou22] Ethereum Foundation. *Fe - The next generation smart contract language for Ethereum*. 2022. URL: https://fe-lang.org/ (visited on 01/14/2022).

[FAH20] Joel Frank, Cornelius Aschermann, and Thorsten Holz. "ETHBMC: A Bounded Model Checker for Smart Contracts". In: *29th USENIX Security Symposium*. USENIX Association, 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/frank.

[GP20] Asem Ghaleb and Karthik Pattabiraman. "How Effective Are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools Using Bug Injection". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA. 2020.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing". In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. Ed. by Vivek Sarkar and Mary W. Hall. PLDI. ACM, 2005. DOI: 10.1145/1065010.1065036.

[GLM12] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. "SAGE: whitebox fuzzing for security testing". In: *Commun. ACM* 55.3 (2012). DOI: 10.1145/2093548.2093564.

[SYZK] Google. *syzkaller - syzkaller is an unsupervised coverage-guided kernel fuzzer*. URL: https://github.com/google/syzkaller (visited on 06/03/2022).

[GMZ20] Rahul Gopinath, Björn Mathis, and Andreas Zeller. "Mining input grammars from dynamic control flow". In: *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2020. DOI: 10.1145/3368089.3409679.

[GRAAL] *GraalVM Reference Manual: Native Image*. URL: https://www.graalvm.org/reference-manual/native-image/ (visited on 09/14/2021).

[Gre+19] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. "Gigahorse: thorough, declarative decompilation of smart contracts". In: *Proceedings of the 41st International Conference on Software Engineering*. Ed. by Joanne M. Atlee, Tevfik Bultan, and Jon Whittle. ICSE. IEEE / ACM, 2019. DOI: 10.1109/ICSE.2019.00120.

[GA22] Harrison Green and Thanassis Avgerinos. "GraphFuzz: Library API Fuzzing with Lifetime-aware Dataflow Graphs". In: *44th IEEE/ACM 44th International Conference on Software Engineering*. ICSE (2022). DOI: 10.1145/3510003.3510228.

[Gri+20] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. "Echidna: effective, usable, and fast fuzzing for smart contracts". In: *29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA. ACM, 2020. DOI: 10.1145/3395363.3404366.

[GMS18] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. "A Semantic Framework for the Security Analysis of Ethereum Smart Contracts". In: *Proceedings of the 7th International Conference on Principles of Security and Trust*. 2018. DOI: 10.1007/978-3-319-89722-6_10.

[Gri+22] Fabio Gritti, Fabio Pagani, Ilya Grishchenko, Lukas Dresel, Nilo Redini, Christopher Kruegel, and Giovanni Vigna. "HEAPSTER: Analyzing the Security of Dynamic Allocators for Monolithic Firmware Images". In: *2022 IEEE Symposium on Security and Privacy*. SP. Apr. 2022. DOI: 10.1109/SP46214.2022.00130.

[GG21] Alex Groce and Gustavo Grieco. "echidna-parade: a tool for diverse multicore smart contract fuzzing". In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA. 2021. DOI: 10.1145/3460319.3469076.

[Gro18] Samuel Groß. "FuzzIL: Coverage Guided Fuzzing for JavaScript Engines". MA thesis. Karlsruhe Institute of Technology, 2018. URL: https://saelo.github.io/papers/thesis.pdf.

[Gro+18] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. "Online detection of effectively callback free objects with applications to smart contracts". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 48:1–48:28. DOI: 10.1145/3158136.

[Gru20] Daniel Gruss. "Transient-Execution Attacks and Defenses". habilitation. Graz University of Technology, June 2020. URL: https://gruss.cc/files/habil.pdf.

[HC17] HyungSeok Han and Sang Kil Cha. "IMF: Inferred Model-based Fuzzer". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. CCS. ACM, 2017. DOI: 10.1145/3133956.3134103.

[HOC19] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. "CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines". In: *26th Annual Network and Distributed System Security Symposium*. NDSS. The Internet Society, 2019. URL: https://www.ndss-symposium.org/ndss-paper/codealchemist-semantics-aware-code-generation-to-find-vulnerabilities-in-javascript-engines/.

[He+19] Jingxuan He, Mislav Balunovic, Nodar Ambroladze, Petar Tsankov, and Martin T. Vechev. "Learning to Fuzz from Symbolic Execution with Application to Smart Contracts". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS. ACM, 2019. DOI: 10.1145/3319535.3363230.

[RDMSA] Aki Helin. *radams: a general-purpose fuzzer*. URL: https://gitlab.com/akihe/radamsa (visited on 05/23/2022).

[HXG] *HexagonToken*. URL: https://etherscan.io/address/0xB5335e24d0aB29C190AB8C2B459238Da1153cEBA.

[Hil+17] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Rosu. *KEVM: A complete semantics of the ethereum virtual machine*. Tech. rep. 2017.

[Hoe+13] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan del Cuvillo. "Using innovative instructions to create trustworthy software solutions". In: *The Second Workshop on Hardware and Architectural Support for Security and Privacy, HASP*. 2013. DOI: 10.1145/2487726.2488370.

[HHZ12] Christian Holler, Kim Herzig, and Andreas Zeller. "Fuzzing with Code Fragments". In: *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*. Ed. by Tadayoshi Kohno. USENIX Association, 2012, pp. 445–458. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler.

[HZ16] Matthias Höschele and Andreas Zeller. "Mining input grammars from dynamic taints". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. Ed. by David Lo, Sven Apel, and Sarfraz Khurshid. ASE. ACM, 2016. DOI: 10.1145/2970276.2970321.

[Hu+15] Hong Hu, Zheng Leong Chua, Zhenkai Liang, and Prateek Saxena. "Identifying Arbitrary Memory Access Vulnerabilities in Privilege-Separated Software". In: *Computer Security - 20th European Symposium on Research in Computer Security, Proceedings, Part II, ESORICS*. 2015. DOI: 10.1007/978-3-319-24177-7_16.

[Hu+16] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks". In: *IEEE Symposium on Security and Privacy (S&P)*. 2016. DOI: 10.1109/SP.2016.62.

[SGXGMP] Intel. *Demo Programs for the GNU* Multiple Precision Arithmetic Library* for Intel® Software Guard Extensions*. URL: https://github.com/intel/sgx-gmp-demo/ (visited on 10/10/2019).

[Intel4] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4*. Order Number 332831-065US. Intel. Dec. 2017.

[Isp+18] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. "Block Oriented Programming: Automating Data-Only Attacks". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2018. DOI: 10.1145/3243734.3243739.

[Isp+20] Kyriakos K. Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. "FuzzGen: Automatic Fuzzer Generation". In: *29th USENIX Security Symposium*. USENIX Association, 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou.

[IA] Jorge Izquierdo and Manuel Araoz. *EIP-897: DelegateProxy*. URL: https://eips.ethereum.org/EIPS/eip-897 (visited on 04/20/2022).

[Jen16] Christoph Jentzsch. *The History of the DAO and Lessons Learned*. Aug. 2016. URL: https://blog.slock.it/the-history-of-the-dao-and-lessons-learned-d06740f8cfa5 (visited on 10/10/2017).

[Jeo+17] Yuseok Jeon, Priyam Biswas, Scott A. Carr, Byoungyoung Lee, and Mathias Payer. "HexType: Efficient Detection of Type Confusion Errors for C++". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2373–2387. DOI: 10.1145/3133956.3134062.

[Jia+16] Yaoqi Jia, Zheng Leong Chua, Hong Hu, Shuo Chen, Prateek Saxena, and Zhenkai Liang. ""The Web/Local" Boundary Is Fuzzy: A Security Study of Chrome's Process-based Sandboxing". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 791–804. DOI: 10.1145/2976749.2978414.

[JLC18] Bo Jiang, Ye Liu, and W. K. Chan. "ContractFuzzer: fuzzing smart contracts for vulnerability detection". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Ed. by Marianne Huchard, Christian Kästner, and Gordon Fraser. ASE. ACM, 2018. DOI: 10.1145/3238147.3238177.

[Jun+21] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. "WINNIE : Fuzzing Windows Applications with Harness Synthesis and Fast Cloning". In: *28th Annual Network and Distributed System Security Symposium*. NDSS. The Internet Society, 2021. URL: https://www.ndss-symposium.org/ndss-paper/winnie-fuzzing-windows-applications-with-harness-synthesis-and-fast-cloning/.

[Jun+20] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. "Stacked borrows: an aliasing model for Rust". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 41:1–41:32. DOI: 10.1145/3371109.

[KLT01] Rauli Kaksonen, Marko Laakso, and Ari Takanen. "Software Security Assessment through Specification Mutations and Fault Injection". In: *Communications and Multimedia Security Issues of the New Century: IFIP TC6 / TC11 Fifth Joint Working Conference on Communications and Multimedia Security (CMS'01)*. 2001. DOI: 10.1007/978-0-387-35413-2_16.

[Kal+18] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. "ZEUS: Analyzing Safety of Smart Contracts". In: *Proceedings 2018 Network and Distributed System Security Symposium (NDSS)*. 2018. DOI: 10.14722/ndss.2018.23082.

[Kin76] James C King. "Symbolic execution and program testing". In: *Commun. ACM* 19.7 (July 1976). ISSN: 0001-0782. DOI: 10.1145/360248.360252.

[KotET] *King of the Ether Throne - Post-Mortem Investigation (Feb 2016)*. 2016. URL: https://www.kingoftheether.com/postmortem.html (visited on 01/14/2022).

[Kle+18] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. "Evaluating Fuzz Testing". In: *ACM Conference on Computer and Communications Security (CCS)*. 2018.

[KR18a] Johannes Krupp and Christian Rossow. "teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts". In: *27th USENIX Security Symposium*. USENIX Association, 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/krupp.

[KR18b] Johannes Krupp and Christian Rossow. "teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts". In: *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/krupp.

[KLS21] Neil Kulkarni, Caroline Lemieux, and Koushik Sen. "Learning Highly Recursive Input Grammars". In: *36th IEEE/ACM International Conference on Automated Software Engineering*. ASE. IEEE, 2021, pp. 456–467. DOI: 10.1109/ASE51524.2021.9678879.

[Kuz+12] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. "Efficient state merging in symbolic execution". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2012. DOI: 10.1145/2254064.2254088.

[Lau+10] M A Laurenzano, M M Tikir, L Carrington, and A Snavely. "PEBIL: Efficient static binary instrumentation for Linux". In: *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. 2010. DOI: 10.1109/ISPASS.2010.5452024.

[Lee+17] Jae-Hyuk Lee, Jin Soo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. "Hacking in Darkness: Return-oriented Programming against Secure Enclaves". In: *26th USENIX Security Symposium, USENIX Security*. 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk.

[LML21] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. "ExpRace: Exploiting Kernel Races through Raising Interrupts". In: *30th USENIX Security Symposium*. USENIX Association, 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/lee-yoochan.

[LZM19] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. "Mimalloc: Free List Sharding in Action". In: *Programming Languages and Systems - 17th Asian Symposium*. APLAS. 2019. DOI: 10.1007/978-3-030-34175-6_13.

[Lim+21] Jungwon Lim, Yonghwi Jin, Mansour Alharthi, Xiaokuan Zhang, Jinho Jung, Rajat Gupta, Kuilin Li, Daehee Jang, and Taesoo Kim. "SOK: On the Analysis of Web Browser Security". In: *CoRR* abs/2112.15561 (2021). arXiv: 2112.15561.

[LLV] LLVM Project. *FuzzedDataProvider.h - Utility header for fuzz targets*. URL: https://github.com/llvm/llvm-project/blob/8b90b2539048a581052a4b0d7628ffba0cd582a9/compiler-rt/include/fuzzer/FuzzedDataProvider.h (visited on 06/03/2022).

[Luk+05] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation". In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. PLDI. 2005. DOI: 10.1145/1065010.1065034.

[Luu+16] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. "Making Smart Contracts Smarter". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS. ACM, 2016. DOI: 10.1145/2976749.2978309.

[Man+21] Valentin J M Manes, Hyungseok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. "The art, science, and engineering of fuzzing: A survey". In: *IEEE Trans. Software Eng.* 47.11 (Nov. 2021), pp. 2312–2331. ISSN: 0098-5589, 1939-3520. DOI: 10.1109/tse.2019.2946563.

[Mar17] Moxie Marlinspike. *Technology preview: Private contact discovery for Signal*. Sept. 26, 2017. URL: https://signal.org/blog/private-contact-discovery/ (visited on 10/10/2019).

[McK+13] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. "Innovative instructions and software model for isolated execution". In: *The Second Workshop on Hardware and Architectural Support for Security and Privacy, HASP*. 2013. DOI: 10.1145/2487726.2488368.

[Mic14] Microsoft Corporation. *Visual Studio 2015 Preview: Work-in-Progress Security Feature*. 2014. URL: http://blogs.msdn.com/b/vcblog/archive/2014/12/08/visual-studio-2015-preview-work-in-progress-security-feature.aspx.

[Mog+20] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. "CopyCat: Controlled Instruction-Level Attacks on Enclaves". In: *29th USENIX Security Symposium*. USENIX Association, 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-copycat.

[Mos+19] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. "Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts". In: *34th IEEE/ACM International Conference on Automated Software Engineering*. ASE. IEEE, 2019. DOI: 10.1109/ASE.2019.00133.

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference (TACAS)*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[MWM] Peter Murray, Nate Welch, and Joe Messerman. *EIP-1167: Minimal Proxy Contract*. URL: https://eips.ethereum.org/EIPS/eip-1167 (visited on 04/20/2022).

[Conc] ConsenSys. *Mythril*. URL: https://github.com/ConsenSys/mythril.

[Nad] Facu Spagnuolo Elena Nadolinski. *ZeppelinOS Blog: Proxy Patterns*. URL: https://blog.zeppelinos.org/proxy-patterns/ (visited on 07/03/2020).

[Nak08] Satoshi Nakamoto. "Bitcoin: A peer-to-peer electronic cash system". In: (2008). URL: http://www.academia.edu/download/32413652/BitCoin_P2P_electronic_cash_system.pdf.

[Ngu+20] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. "sFuzz: an efficient adaptive fuzzer for solidity smart contracts". In: *ICSE '20: 42nd International Conference on Software Engineering*. Ed. by Gregg Rothermel and Doo-Hwan Bae. ACM, 2020. DOI: 10.1145/3377811.3380334.

[Nik+18] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. "Finding The Greedy, Prodigal, and Suicidal Contracts at Scale". In: *Proceedings of the 34th Annual Computer Security Applications Conference*. ACSAC. 2018. DOI: 10.1145/3274694.3274743.

[One96] Aleph One. *Smashing The Stack For Fun And Profit*. Nov. 1996. URL: http://phrack.org/issues/49/14.html.

[REGu] *OpenZeppelin Documentation: ReentrancyGuard*. URL: https://docs.openzeppelin.com/contracts/4.x/api/security#ReentrancyGuard (visited on 08/15/2022).

[Paa+21b] David Paaßen, Sebastian Surminski, Michael Rodler, and Lucas Davi. "My Fuzzer Beats Them All! Developing a Framework for Fair Evaluation and Comparison of Fuzzers". In: *Proc. of European Symposium on Research in Computer Security (ESORICS)*. Springer International Publishing, 2021.

[PaX] PaX Team. *PaX: PAGEEXEC Design*. URL: https://pax.grsecurity.net/docs/pageexec.txt (visited on 08/23/2019).

[PBG+13] Mathias Payer, Boris Bluntschli, Thomas R Gross, et al. "DynSec: On-the-fly Code Rewriting and Repair". In: *HotSWUp*. 2013. URL: http://bitblaze.cs.berkeley.edu/papers/payer13dynsec.pdf.

[Pec18] PeckShield. *New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706)*. 2018. URL: https://blog.peckshield.com/2018/05/10/multiOverflow/ (visited on 05/27/2019).

[Peca] PeckShield. *ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299)*. URL: https://blog.peckshield.com/2018/04/22/batchOverflow/ (visited on 05/27/2019).

[Pecb] PeckShield. *New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239)*. URL: https://blog.peckshield.com/2018/05/18/burnOverflow/ (visited on 05/27/2019).

[Pecc] PeckShield. *PeckShield advisories*. URL: https://blog.peckshield.com/advisories.html (visited on 05/27/2019).

[PL21] Daniel Perez and Benjamin Livshits. "Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited". In: *30th USENIX Security Symposium*. USENIX Association, 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/perez.

[PKH19] Jannik Pewny, Philipp Koppe, and Thorsten Holz. "STEROIDS for DOPed Applications: A Compiler for Automated Data-Oriented Programming". In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. 2019. DOI: 10.1109/EuroSP.2019.00018.

[PF20] Sebastian Poeplau and Aurélien Francillon. "Symbolic execution with SymCC: Don't interpret, compile!" In: *29th USENIX Security Symposium*. USENIX Association, 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau.

[PF21] Sebastian Poeplau and Aurélien Francillon. "SymQEMU: Compilation-based symbolic execution for binaries". In: *28th Annual Network and Distributed System Security Symposium*. NDSS. The Internet Society, 2021. URL: https://www.ndss-symposium.org/ndss-paper/symqemu-compilation-based-symbolic-execution-for-binaries/.

[Pop19] Gregory Popovitch. *The Parallel Hashmap*. Mar. 2019. URL: https://greg7mdp.github.io/parallel-hashmap/ (visited on 04/22/2022).

[Pri16] Rob Price. *Digital currency Ethereum is cratering because of a $50 million hack*. June 2016. URL: https://www.businessinsider.com/dao-hacked-ethereum-crashing-in-value-tens-of-millions-allegedly-stolen-2016-6 (visited on 04/20/2018).

[Raw+17] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. "VUzzer: Application-aware Evolutionary Fuzzing". In: *24th Annual Network and Distributed System Security Symposium*. NDSS. The Internet Society, 2017. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/.

[Rob+] Robert Seacord, Jill Britton et al. *SEI CERT C Coding Standard: INT30-C. Ensure that unsigned integer operations do not wrap*. URL: https://wiki.sei.cmu.edu/confluence/display/c/INT30-C.+Ensure+that+unsigned+integer+operations+do+not+wrap (visited on 08/16/2022).

[Rod] Michael Rodler. *Re-Entrancy Attack Patterns*. URL: https://github.com/uni-due-syssec/eth-reentrancy-attack-patterns (visited on 08/05/2022).

[Rud07] Jesse Ruderman. *Introducing jsfunfuzz*. 2007. URL: https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/ (visited on 06/03/2022).

[OZSM] *SafeMath library - Wrappers over Solidity's arithmetic operations with added overflow checks*. URL: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol (visited on 06/03/2020).

[Sch+20a] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. *SGAxe: How SGX Fails in Practice*. https://sgaxeattack.com/. 2020.

[Sch+20b] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. "eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts". In: *ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2020. DOI: 10.1145/3372297.3417250.

[SSM20] Clara Schneidewind, Markus Scherer, and Matteo Maffei. "The Good, The Bad and The Ugly: Pitfalls and Best Practices in Automated Sound Static Analysis of Ethereum Smart Contracts". In: *Leveraging Applications of Formal Methods, Verification and Validation: Applications*. Springer International Publishing, 2020. DOI: 10.1007/978-3-030-61467-6_14.

[Sch+16] Bernhard Scholz, Herbert Jordan, Pavle Subotic, and Till Westmann. "On fast large-scale program analysis in Datalog". In: *Proceedings of the 25th International Conference on Compiler Construction*. CC. 2016. DOI: 10.1145/2892208.2892226.

[Sch+21] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. "Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types". In: *30th USENIX Security Symposium*. USENIX Association, Aug. 2021. ISBN: 978-193-913-3-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo.

[Sch+22] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. "Nyx-net: network fuzzing with incremental snapshots". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. EuroSys. Association for Computing Machinery, Mar. 2022. ISBN: 978-145-039-1-6-2-7. DOI: 10.1145/3492321.3519591.

[Sch+15a] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. "VC3: Trustworthy Data Analytics in the Cloud Using SGX". In: *2015 IEEE Symposium on Security and Privacy, S&P*. 2015. DOI: 10.1109/SP.2015.10.

[Sch+15b] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Aymad-Reza Sadeghi, and Thorsten Holz. "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications". In: *2015 IEEE Symposium on Security and Privacy (S&P)*. 2015. DOI: 10.1109/SP.2015.51.

[SAB10] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)". In: *31st IEEE Symposium on Security and Privacy, S&P*. 2010. DOI: 10.1109/SP.2010.26.

[Sch+18] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. "Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features". In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. ACM, May 2018. DOI: 10.1145/3196494.3196508.

[Sch+17] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Malware Guard Extension: Using SGX to Conceal Cache Attacks". In: *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference*. DIMVA. Springer, 2017. DOI: 10.1007/978-3-319-60876-1_1.

[See] Donn Seeley. *A Tour of the Worm*. URL: https://web.archive.org/web/20070520233435/http://world.std.com/~franl/worm.html (visited on 11/02/2007).

[Seo+17] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs". In: *24th Annual Network and Distributed System Security Symposium, NDSS*. 2017. DOI: 10.14722/ndss.2017.23037.

[Ser+12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. "Address-Sanitizer: A Fast Address Sanity Checker". In: *USENIX Annual Technical Conference*. 2012. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[Sha07] Hovav Shacham. "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)". In: *ACM Conference on Computer and Communications Security (CCS)*. 2007. DOI: 10.1145/1315245.1315313.

[Sho+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *IEEE Symposium on Security and Privacy, S&P*. 2016. DOI: 10.1109/SP.2016.17.

[Sim] Simon Johnson, Raghunandan Makaram, Amy Santoni, Vinnie Scarlata. *Supporting Intel® SGX on Multi-Socket Platforms*. Tech. rep. Intel Corporation. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/supporting-intel-sgx-on-mulit-socket-platforms.pdf.

[SliU] *Slither Wiki: Upgradeability Checks*. URL: https://github.com/crytic/slither/wiki/Upgradeability-Checks (visited on 06/02/2019).

[SWC] *Smart Contract Weakness Classification and Test Cases*. 2022. URL: https://swcregistry.io/ (visited on 01/14/2022).

[SMT] *SmartMeshICO*. URL: https://etherscan.io/address/0x55F93985431Fc9304077687a35A1BA103dC1e081.

[SHO21] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. "SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution". In: *30th USENIX Security Symposium*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/so.

[So+20] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. "VERISMART: A Highly Precise Safety Verifier for Ethereum Smart Contracts". In: *2020 IEEE Symposium on Security and Privacy*. SP. IEEE, 2020. DOI: 10.1109/SP40000.2020.00032.

[SCA] *Social Chain*. URL: https://etherscan.io/address/0xb75a5e36cc668bc8fe468e8f272cd4a0fd0fd773.

[SolRe] *Solidity by Example: Re-Entrancy*. URL: https://solidity-by-example.org/hacks/re-entrancy/ (visited on 12/12/2021).

[Sol080] *Solidity Changelog 0.8.0*. URL: https://github.com/ethereum/solidity/blob/develop/Changelog.md#080-2020-12-16 (visited on 04/20/2022).

[Sol] Solidity Developers. *Solidity Documentation: Layout of State Variables in Storage*. URL: https://solidity.readthedocs.io/en/v0.5.10/miscellaneous.html (visited on 05/27/2019).

[SolSH] *Solidity Documentation: Encoding of the Metadata Hash in the Bytecode*. URL: https://solidity.readthedocs.io/en/v0.5.8/metadata.html#encoding-of-the-metadata-hash-in-the-bytecode (visited on 05/27/2018).

[SolWd] *Solidity Withdrawal from Contracts*. URL: https://solidity.readthedocs.io/en/develop/common-patterns.html#withdrawal-from-contracts (visited on 07/25/2018).

[SolCEI] *Solidity: Security Considerations - Use the Checks-Effects-Interactions Pattern*. URL: https://docs.soliditylang.org/en/v0.8.7/security-considerations.html#use-the-checks-effects-interactions-pattern (visited on 09/09/2021).

[Spa18] SpankChain. *SpankChain: We Got Spanked: What We Know So Far*. Oct. 2018. URL: https://medium.com/spankchain/we-got-spanked-what-we-know-so-far-d5ed3a0f38fe (visited on 01/14/2022).

[Ste+16] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution". In: *NDSS*. Vol. 16. 2016.

[Sui17] Matt Suiche. *Porosity: A Decompiler for Blockchain-Based Smart Contract Bytecode*. 2017. URL: https://github.com/comaeio/porosity.

[Sze+13] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal War in Memory". In: *2013 IEEE Symposium on Security and Privacy, S&P*. 2013. DOI: 10.1109/SP.2013.13.

[Tea18] Solidity Team. *Solidity Documentation*. 2018. URL: http://solidity.readthedocs.io/ (visited on 08/06/2018).

[Tea22a] Solidity Team. *Solidity Documentation - Contract ABI Specification*. 2022. URL: https://docs.soliditylang.org/en/v0.8.11/abi-spec.html (visited on 01/14/2022).

[Tea22b] Solidity Team. *Solidity Version 0.8.11*. 2022. URL: https://github.com/ethereum/solidity/releases/tag/v0.8.11 (visited on 01/14/2022).

[Tec17a] Parity Technologies. *A Postmortem on the Parity Multi-Sig Library Self-Destruct*. http://paritytech.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/. [Online; accessed Jun 3, 2020]. Nov. 2017.

[Tec17b] Parity Technologies. *Security Alert - Parity Wallet*. http://paritytech.io/security-alert. [Online; accessed 6-April-2018]. Nov. 2017.

[dDAO] *TheDarkDAO contract address*. URL: https://etherscan.io/address/0x304a554a310C7e546dfe434669C62820b7D83490 (visited on 08/01/2018).

[Tho19] Gavin Thomas. July 2019. URL: https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/ (visited on 09/21/2020).

[Thu+11] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. "Synthesizing method sequences for high-coverage testing". In: *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA. 2011. DOI: 10.1145/2048066.2048083.

[Tic+14] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM". In: *23rd USENIX Security Symposium*. USENIX Association, 2014. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice.

[Tor+21a] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. "ConFuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts". In: *IEEE European Symposium on Security and Privacy*. EuroS&P. IEEE, 2021. DOI: 10.1109/EuroSP51992.2021.00018.

[Tor+21b] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. "The Eye of Horus: Spotting and Analyzing Attacks on Ethereum Smart Contracts". In: *Financial Cryptography and Data Security - 25th International Conference*. FC. Springer, 2021. DOI: 10.1007/978-3-662-64322-8_2.

[TJS22] Christof Ferreira Torres, Hugo Jonker, and Radu State. "Elysium: Automagically Healing Vulnerable Smart Contracts Using Context-Aware Patching". In: (2022).

[TSS19] Christof Ferreira Torres, Mathis Steichen, and Radu State. "The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts". In: *28th USENIX Security Symposium*. 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/ferreira.

[Tra18] Trail of Bits. *Trail of Bits Blog: How contract migration works*. 2018. URL: https://blog.trailofbits.com/2018/10/29/how-contract-migration-works/ (visited on 06/03/2020).

[TPV17] Chia-che Tsai, Donald E. Porter, and Mona Vij. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX". In: *2017 USENIX Annual Technical Conference, USENIX ATC*. 2017. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

[Tsa+18] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. "Securify: Practical security analysis of smart contracts". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS. ACM, 2018.

[il2cpp] *Unity Documentation: IL2CPP Overview*. URL: https://docs.unity3d.com/Manual/IL2CPP.html (visited on 09/14/2021).

[UET] *UselessEthereumToken*. URL: https://etherscan.io/address/0x27f706edde3aD952EF647Dd67E24e38CD0803DD6.

[Van+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution". In: *27th USENIX Security Symposium*. 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[Van+19] Jo Van Bulck, David F. Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. "A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS. ACM, 2019. DOI: 10.1145/3319535.3363206.

[VPS17] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: *Proceedings of the 2Nd Workshop on System Software for Trusted Execution, SysTEX*. 2017. DOI: 10.1145/3152701.3152706.

[Vee+17] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrida. "The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS*. 2017. DOI: 10.1145/3133956.3134026.

[Vyp] *Vyper*. URL: https://github.com/ethereum/vyper.

[Wan+19a] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. "Towards Memory Safe Enclave Programming with Rust-SGX". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS*. 2019. DOI: 10.1145/3319535.3354241.

[Wan+19b] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. "Superion: grammar-aware greybox fuzzing". In: *Proceedings of the 41st International Conference on Software Engineering*. IEEE / ACM, 2019. DOI: 10.1109/ICSE.2019.00081.

[Wan+17] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. "How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel". In: *26th USENIX Security Symposium*. USENIX Association, Aug. 2017. ISBN: 978-193-197-1-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-pengfei.

[Wil16] Jeffrey Wilcke. https://blog.ethereum.org/2016/09/22/ethereum-network-currently-undergoing-dos-attack/. [Online; accessed Jul 28, 2018]. Sept. 2016.

[Wim+19] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. "Initialize once, start fast: application initialization at build time". In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019). DOI: 10.1145/3360610.

[WOLFex] *wolfSSL Linux Enclave Example*. URL: https://github.com/wolfSSL/wolfssl-examples/tree/master/SGX_Linux (visited on 10/10/2019).

[WOLF] *wolfSSL: a small, fast, portable implementation of TLS/SSL for embedded devices to the cloud*. Oct. 10, 2019. URL: https://github.com/wolfSSL/wolfssl (visited on 08/27/2019).

[Woo19] Gavin Wood. *Ethereum: A secure decentralised generalised transaction ledger*. Specification. Version BYZANTIUM VERSION 7e819ec - 2019-10-20. Ethereum Foundation, Oct. 20, 2019. URL: https://ethereum.github.io/yellowpaper/paper.pdf.

[WC20] Valentin Wüstholz and Maria Christakis. "Harvey: a greybox fuzzer for smart contracts". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. ACM, Nov. 2020. DOI: 10.1145/3368089.3417064.

[Xie+05] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. "Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution". In: *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference*. TACAS. 2005. DOI: 10.1007/978-3-540-31980-1_24.

[Xu+18] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. "Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels". In: *2018 IEEE Symposium on Security and Privacy*. SP. IEEE, May 2018. DOI: 10.1109/SP.2018.00017.

[Xu+17] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. "Designing New Operating Primitives to Improve Fuzzing Performance". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS. 2017. DOI: 10.1145/3133956.3134046.

[XCP15] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". In: *2015 IEEE Symposium on Security and Privacy*. S&P. IEEE Computer Society, 2015. DOI: 10.1109/SP.2015.45.

[Yan+11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and understanding bugs in C compilers". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, June 2011. DOI: 10.1145/1993498.1993532.

[Yun+18] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. "QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing". In: *27th USENIX Security Symposium*. USENIX Association, 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/yun.

[Zal14] Michal Zalewski. *Pulling JPEGs out of thin air*. 2014. URL: https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html (visited on 05/23/2022).

[Zal] Michal Zalewski. *American Fuzzy Lop*. URL: https://lcamtuf.coredump.cx/afl/ (visited on 04/28/2022).

[ZepOS] *ZeppelinOS Documentation*. URL: https://docs.zeppelinos.org/ (visited on 08/16/2022).

[Zer] Google Project Zero. *0-days In-the-Wild*. URL: https://googleprojectzero.github.io/0days-in-the-wild/ (visited on 01/18/2022).

[Zha+20] Yuyao Zhang, Siqi Ma, Juanru Li, Kailai Li, Surya Nepal, and Dawu Gu. "SMARTSHIELD: Automatic Smart Contract Protection Made Easy". In: *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, (SANER)*. 2020. DOI: 10.1109/SANER48275.2020.9054825.

[Zho+20] Shunfan Zhou, Zhemin Yang, Jie Xiang, Yinzhi Cao, Min Yang, and Yuan Zhang. "An Ever-evolving Game: Evaluation of Real-world Attacks and Defenses in Ethereum Ecosystem". In: *29th USENIX Security Symposium*. USENIX Association, 2020.

[Zho+18] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. "Erays: Reverse Engineering Ethereum's Opaque Smart Contracts". In: *27th USENIX Security Symposium*. USENIX Association, 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/zhou.

LIST OF ACRONYMS

ABI application binary interface. 32, 37, 49, 72, 100, 101, 110

API application programming interface. 9, 17, 18, 22–24, 29, 32, 45, 48, 70–72, 74, 75, 83

ASLR address space layout randomization. 67

CFG control-flow graph. 37, 43, 162, 163, 171, 172, 174, 192

CFI control-flow integrity. 34, 73

CPS cyber-physical system. 1, 5

CPU central processing unit. 2, 3, 16, 26–28, 31, 33, 34, 38, 50, 53, 162, 195

DApp decentralized application. 2

DeFI decentralized finance. 1, 3

DOP data-oriented programming. 47

EDL enclave definition language. 29, 30

ELF Executable and Linkable Format. 53

EOA externally-owned account. 33, 34, 36, 37

EPC enclave page cache. 27, 28

EVM Ethereum virtual machine. 7, 26, 34–39, 43, 44, 96, 97, 128, 131, 133–137, 139, 140, 142, 148–150, 154, 156, 161–167, 171, 172, 174, 175, 178, 182, 186, 187, 192, 197

I/O input/output. 29

ISA instruction-set architecture. 25, 31, 34, 37, 38, 53, 162

IT Information Technology. 1

JIT just-in-time. 22

OS operating system. 15, 19, 27, 31, 50, 61, 62, 67, 70, 76

OT Operational Technology. 1

PC program counter. 35

PoC proof-of-concept. 48, 67, 69, 72–74, 77, 79, 81, 82

ROP return-oriented programming. 32, 34, 47, 67, 69

SGX Intel software guard extensions. iii, v, 2–6, 9, 10, 17, 25–29, 31, 32, 45, 47–50, 52–56, 58, 61, 62, 65–67, 69, 70, 72, 76, 77, 79, 81, 82, 195, 196

SMT satisfiability modulo theories. 18

TCB trusted computing base. 1, 2, 26, 81, 196

TCS Thread Control Structure. 27, 62

TEE trusted execution environment. 2, 4, 7, 25, 196

TOCTOU time-of-check time-of-use. 14, 15, 57, 62

TRTS trusted runtime. 29, 52, 53

UAF use-after-free. 14, 64

URTS untrusted runtime. 29

vtable virtual method table. 72–74

Eidesstattliche Erklärung zu § 14 Abs. 1 Nr. 6 der Promotionsordnung

Ich gebe folgende eidesstattliche Erklärung ab:

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig ohne unzulässige Hilfe Dritter verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und alle wörtlich oder inhaltlich übernommenen Stellen unter der Angabe der Quelle als solche gekennzeichnet habe.

Die Grundsätze für die Sicherung guter wissenschaftlicher Praxis an der Universität Duisburg-Essen sind beachtet worden.

Ich habe die Arbeit keiner anderen Stelle zu Prüfungszwecken vorgelegt.

Essen, December 5, 2023 Unterschrift