UNIVERSITÄT DUISBURG ESSEN

Open-Minded

Securing Embedded Devices with Remote Attestation

Dissertation
zur Erlangung des Doktorgrades

„Dr. rer. nat."

der Fakultät für Wirtschaftswissenschaften
der Universität Duisburg-Essen

vorgelegt von

Sebastian Erasmus Raphael Josef Surminski

aus
Aachen, Deutschland

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

Essen, Juni 2024

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

2. Gutachter:
Prof. Dr. Ahmad-Reza Sadeghi
Technische Universität Darmstadt

Tag der mündlichen Prüfung: 14. März 2024

Securing Embedded Devices with Remote Attestation
Sebastian Erasmus Raphael Josef Surminski
sebastian@surminski.org
`https://www.surminski.org/`

Abstract

Embedded devices are ubiquitous in modern society. They are critical components in smart factories, vehicles, critical infrastructures, and medical devices. Recent studies and reports have revealed that many of these devices suffer from crucial vulnerabilities that can be exploited with fatal consequences. Despite their safety- and security-critical roles, these devices often do not feature state-of-the-art security mechanisms. Moreover, the machines equipped with these devices have a long lifetime, and changing hardware components to integrate hardware-based security solutions is often not possible as these embedded devices are deeply integrated into other systems. Systems operating under real-time constraints are especially critical. Real-time systems have strict timing requirements, and integrating new security mechanisms is not a viable option as they often influence the device's runtime behavior. One solution is to offload security enhancements to a remote instance, for example using remote attestation. Remote attestation is a powerful security service for validating the trustworthiness of a remote device. However, implementing remote attestation in existing legacy devices is a challenging task: How to obtain a trustworthy self-measurement of a device, even if the system is fully compromised?

This dissertation presents different remote attestation solutions for specific classes of embedded devices. RealSWATT is the first software-based remote-attestation scheme for embedded systems with real-time constraints. With SCAtt-man, this dissertation introduces an attestation solution for consumer IoT devices, that allows user-observable attestation, and thereby also solves the problem of missing device authentication in software-based attestation. Furthermore, this dissertation presents an external attestation device called DMA'n'Play that leverages direct memory access (DMA) to monitor the memory content of the attested device. As DMA is independent of the main processor, this solution is also applicable to devices with real-time constraints.

However, remote attestation does not prevent the compromise of the attested system. Therefore, it is eventually necessary to patch vulnerabilities. However, patching embedded devices typically influences the operation of the patched device. Often, even a reboot is required. For many critical embedded devices, especially such with real-time constraints, rebooting is mostly not possible, preventing the timely application of patches. To tackle this problem, this dissertation introduces HERA, a framework for patching devices without interfering with normal operation, even on systems with real-time tasks.

DOI: 10.17185/duepublico/82079
URN: urn:nbn:de:hbz:465-20240619-112310-6

Zusammenfassung

Eingebettete Geräte sind in der modernen Gesellschaft allgegenwärtig. Sie sind wichtige Komponenten in Fabriken, Fahrzeugen, kritischer Infrastruktur und medizinischen Geräten. Jüngste Studien und Berichte haben gezeigt, dass viele dieser Geräte Schwachstellen aufweisen, die mit fatalen Folgen ausgenutzt werden können. Trotz ihrer Sicherheitsrelevanz verfügen diese Geräte oft nicht über moderne Sicherheitsmechanismen, die eine Ausnutzung verhindern würden. Ein Austausch der Geräte ist zumeist nicht praktikabel, da diese oftmals tief in andere Systeme integriert sind. Besonders kritisch sind Echtzeitsysteme, die strenge Zeitvorgaben bei der Ausführung ihrer Aufgaben haben. Neue Sicherheitsmechanismen beeinflussen jedoch oft das Laufzeitverhalten des Geräts. Eine Lösung besteht darin, diese Sicherheitsverbesserungen auf eine entfernte Instanz auszulagern, zum Beispiel durch Remote Attestation. Remote Attestation ist ein Verfahren, das es erlaubt die Vertrauenswürdigkeit eines entfernten Geräts zu überprüfen. Allerdings ist die Implementierung eines solchen Verfahrens in bestehende Altgeräte eine komplexe Aufgabe: Wie erhält man eine vertrauenswürdige Messung eines nicht vertrauenswürdigen und potentiell kompromittierten Gerätes?

Diese Dissertation stellt neue Sicherheitslösungen für verschiedene Arten von eingebetteten Geräte vor. RealSWATT ist das erste Verfahren für softwarebasierte Remote Attestation für eingebettete Systeme mit Echtzeitanforderungen. Mit SCAttman wird eine Lösung für Remote Attestation von IoT-Geräten für Endkunden entwickelt, die eine vom Benutzer beobachtbare Attestation ermöglicht. Darüber hinaus stellt diese Dissertation ein externes Gerät zur Unterstützung der Attestation namens DMA'n'Play vor, das Direct Memory Access (DMA) nutzt, um den Speicherinhalt des attestierten Geräts zu überwachen. Da DMA unabhängig vom Prozessor erfolgt, ist diese Lösung auch für Geräte mit Echtzeitanforderungen geeignet. Remote Attestation verhindert jedoch nicht die Kompromittierung des attestierten Systems. Daher ist es letztendlich notwendig, Sicherheitslücken zu schließen. Das Patchen hat jedoch in der Regel Auswirkungen auf den Betrieb des gepatchten Geräts, oft ist sogar ein Neustart erforderlich. Für viele kritische eingebettete Geräte ist dies jedoch meist nicht möglich, was die zeitnahe Anwendung von Patches verhindert. Um dieses Problem zu lösen, wurde im Rahmen dieser Dissertation das Framework HERA entwickelt, das ein Patchen von eingebetteten Systemen mit Echtzeitanforderungen während des Betriebs erlaubt, ohne dabei Einfluss auf das Laufzeitverhalten zu nehmen.

Acknowledgements

First, I would like to thank my advisor Lucas Davi for giving me the opportunity to pursue a PhD, showing me how to convert my ideas into actual research, and bringing this research in a form that people actually understand. Over the years, I had the pleasure of collaborating with various people. I want to thank Ferdinand Brasser, Tobias Cloosters, Kilian Demuth, Christian Niesler, Sebastian Linsner, David Paaßen, Christian Reuter, Michael Rodler, and Ahmad-Reza Sadeghi for their contribution and input to various research topics we pursued. In particular, I want to thank Ahmad-Reza Sadeghi for being the second referee of my dissertation.

In addition, I would like to thank my colleagues, Michael, Tobias, Jens-Rene, David, Oussama, Shahid, Hagen, Shivam, and Christian, for not only inspiring but also fun discussions during coffee and lunch breaks. I want to especially thank Michael Rodler, who always had an open ear and honest feedback when it came to tricky questions. It was a pleasure working with all of you!

Finally, I want to thank my wife, Sarah Wedrich, my friends, and my family. Thank you, Sarah, for supporting me with everything I do and tolerating that I often invested far too many hours into my work, but also showed me how to manage my time and projects successfully.

Contents

List of Figures

List of Tables

Listings

Acronyms

API	Application Programming Interface
ASLR	Address Space Layout Randomization
ATI	Affinity for Technology Interaction
BL	Branch-and-Link
CFI	Control Flow Integrity
CoAP	Constrained Application Protocol
COTS	Commercial off-the-Shelf
CPU	Central Processing Unit
CVSS	Common Vulnerability Scoring System
DDoS	Distributed Denial-of-Service
DMA	Direct Memory Access
DoS	Denial-of-Service
ELF	Executable and Linkable Format
FDA	Food and Drug Administration
FFT	Fast Fourier Transform
FPB	Flash Patch and Breakpoint
GPIO	General-Purpose Input/Output
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
ICS	Industrial Control System
IEEE	Institute of Electrical and Electronics Engineers
IIoT	Industrial Internet of Things
IOMMU	Input/Output Memory Management Unit
IoT	Internet of Things
IP	Internet Protocol
ISA	Instruction Set Architecture
IT	Information Technology
ITU	International Telecommunication Union
LED	Light-Emitting Diode
LR	Link Register
MISO	Master Input, Slave Output
MitM	Man-in-the-Middle
MMU	Memory Management Unit
MOSI	Master Out, Slave In

MPU	Memory Protection Unit
NIST	National Institute of Standards and Technology
NOP	No Operation
LLC	Last Level Cache
OTA	Over-the-Air
OS	Operating System
OWASP	Open Worldwide Application Security Project
PCI	Peripheral Component Interconnect
PLC	Programmable Logic Controller
PUF	Physically Unclonable Function
PoC	Proof-of-Concept
RAM	Random Access Memory
RA	Remote Attestation
REST	Representational State Transfer
RGB	Red, Green, Blue
ROM	Read-Only Memory
ROP	Return-Oriented Programming
RTOS	Real-Time Operating System
RX	Reception
SGX	Software Guard Extensions
SHA	Secure Hash Algorithm
SoC	System-on-Chip
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
SS	Slave Select
STT	Speech-to-Text
SVC	Super Visor Call
SWATT	Software-Based Attestation
TCP	Transmission Control Protocol
TEE	Trusted Execution Environment
TLS	Transport Layer Security
TOCTOU	Time-of-Check/Time-of-Use
TPM	Trusted Platform Module
TTS	Text-to-Speech
TX	Transmission
UART	Universal Asynchronous Receiver-Transmitter
UDP	User Datagram Protocol
UEQ	User Experience Questionnaire
USART	Universal Synchronous and Asynchronous Receiver-Transmitter
UX	User Experience

CHAPTER 1

Introduction

In today's world, embedded systems are everywhere, so to say, ubiquitous. They are crucial parts in many devices, measuring, controlling, and connecting various actuators, sensors, and other components, enabling a wide range of functionality. Embedded devices serve manifold safety-critical tasks in smart factories, cars, medical devices, and critical infrastructure. Integrated into cyber-physical systems, embedded devices often operate under strict real-time constraints, as these systems interact with the physical world, for example, in vehicles or industrial robots [161, 218]. Embedded devices are also widely used to enhance previously unconnected devices with Internet access, connecting consumer devices to the so-called Internet of Things (IoT). In addition, industrial machines and components are also connected via the Internet, resulting in the Industrial Internet of Things (IIoT).

Despite their criticality, embedded devices suffer from various security vulnerabilities [12, 74, 76, 208], including industrial robots [218], vehicles [161], and drones [13]. Typical IoT devices are black-box systems to the user with a limited understanding of threats to security and privacy [299]. In a recent study, 48% of companies reported that they are unable to detect whether an IoT device on their network suffers from a breach, and, for example, is part of a botnet [119]. IoT & Edge Developer Survey by the Eclipse Foundation finds that security is the top concern of developers. 46% of IoT & Edge developers report security concerns, more than connectivity (38%), and deployment (31%) [88].

Numerous examples show that a compromise or malfunction of such devices can have severe implications and even cause physical damage in the real world. The consequences are manifold and reach from the targeted compromise of specific devices to broad large-scale cyberattacks, as several prominent incidents in recent years illustrate. Stuxnet [106] is a well-known example of an attack on industrial control systems targeted at a uranium enrichment plant. Eventually, this attack not only physically damaged centrifuges in a long-term process by altering the power supply of the centrifuges [107], but also caused about 100,000 infections worldwide [169]. Another example is the attack on the control systems of a German steel mill that caused the blast furnace to not shut down properly, leading to severe physical damage [302]. But also, systems outside of the industry are vulnerable. Miller and Valasek remotely controlled an unaltered Jeep Cherokee by exploiting a vulnerability in the head unit [185]. These examples illustrate how dangerous attacks against embedded devices can be.

In contrast to these targeted attacks, embedded devices can also be used for large-scale attacks. The Mirai botnet [21] specifically targeted consumer IoT devices, compromising and controlling millions of IoT devices. This botnet was used for multiple large-scale distributed denial-of-service attacks. It has even been accounted to have performed some of the largest denial-of-service attacks to date, targeting the OVH hoster and many popular websites. These attacks with a bandwidth up to 1.1 Tbit/s involved more than 300,000 devices [125]. This problem is further amplified as embedded devices often lack basic security mechanisms that are common in most other types of systems [12]. For example, more than 80% of the embedded devices do not feature standard security mitigations such as address space layout randomization (ASLR), non-executable memory, and stack canaries [284].

Integrating new security mechanisms into such embedded devices is a challenging task: Legacy real-time embedded devices lack hardware security features, for example, secure boot or trusted execution environments (TEEs). Moreover, they are commonly integrated into other devices, such as machines, control units, and custom circuit boards, and run customized software. Hence, they cannot be simply replaced or upgraded. In contrast to commodity computer systems, in embedded devices, both hardware and software are specifically tailored to their use case, making adjustments difficult. Furthermore, these devices often have a long lifetime. Therefore, there are many legacy devices. For instance, industrial robots have a lifetime of ten years [8, 37], cars in the US are on average 12.1 years old [53], and airplanes have a design lifetime of more than 30 years [7].

In addition, many of these devices also perform real-time or safety-critical tasks. During development, the correctness of functionality and timing behavior has been ensured [79, 277]. However, incorporating protection mechanisms in software such as control flow integrity (CFI) [1] always impacts the execution times of tasks [79, 277]. This is highly critical in the real-time realm since they must adhere to strict timing behavior and hence go through an extensive development and profiling phase. Any changes in execution, even through instrumentation, for example, to integrate control flow integrity [79, 277], or abnormality monitoring [246], affect the runtime behavior of the device, and hence require repeating these extensive testing routines. These circumstances hinder the integration of new security mechanisms into existing legacy devices.

Nevertheless, as eluded before, these embedded devices often have vulnerabilities that can be exploited in practice. This requires security solutions that can be integrated into such legacy devices. A solution is to move security checks to another entity so that the integration of new security mechanisms, such as the detection of malware injection and data manipulation, does not require any changes to the original device.

One way to do so is remote attestation, as it allows an external entity to monitor and attest the internal behavior and state of a remote device [71]. Remote attestation enables a device, the so-called verifier, to check the integrity of another device, the prover. So, the verifier can monitor the operation of the attested device, thereby not only detecting compromises but also enhancing trust in the attested device. This feature is particularly relevant for embedded devices, which often operate in untrusted environments. Remote attestation does not only allow the implementation of further security checks but by design also enables monitoring of attested devices. Monitoring is complementary to further security checks: While additional security checks enhance the security of a device, they do not allow the inspection of a device's state. New security features such as control flow integrity (CFI) can still be circumvented, allowing the compromise of a device. Without monitoring, this manipulation will remain undetected. Remote attestation, in contrast, allows one to determine the state of a system and to detect a compromise even though the compromise is based on a novel attack technique, a zero-day vulnerability, or a backdoor. So remote attestation is not only a worthwhile solution to enhance the security of individual devices but is especially suitable for centrally managed devices that are already monitored, for instance in companies and factories.

The main challenge in remote attestation is to perform a trustworthy self-measurement of an untrusted device: even on a fully compromised system, an attacker may not be able to alter the attestation self-measurement. Many different remote attestation schemes have been proposed to tackle this problem [3, 4, 49, 57, 201, 241]. These approaches nevertheless have different complex demands that hinder their practical usage: Hardware-based approaches require trusted computing modules such as ARM TrustZone to perform secure measurements of the attested device [3, 4], often unavailable on small and embedded devices. Software-based approaches are based on precise measurements of execution time and therefore have strict requirements towards their implementation and communication, limiting their practical applicability [57, 241]. Hybrid approaches need custom hardware extensions that are expensive for initial implementation and are not available on legacy devices [49, 201].

In summary, remote attestation is a possibility to outsource security checks to external devices, thereby enhancing the security of legacy devices whose hardware cannot be changed. This leverages the need for remote attestation solutions that neither require additional hardware nor trusted computing components on the device itself and work on a large variety of embedded devices, including those with real-time capabilities. In this thesis, we develop such attestation schemes and other security enhancements for different types of embedded devices, focusing on the particular requirements of legacy devices, systems with real-time constraints, and IoT devices.

1.1 Goal and Scope of this Dissertation

The main research question in this thesis is ‘How can we secure embedded legacy devices?’. Considering the large number of embedded devices already deployed, security enhancements for these legacy devices are an important issue. Targeted attacks on industrial control systems and vehicles illustrated the criticality of embedded devices. However, a broad compromise of individually uncritical IoT devices can also have severe consequences, as the Mirai botnet showed. It used large numbers of compromised IoT devices for distributed denial-of-service attacks, causing some of the largest denial-of-service attacks to date. In contrast to commodity computer systems, embedded systems have specific characteristics that hinder changes and updates to their software: Legacy embedded devices are specifically designed for their use case and have strict hardware limitations. This results in a large variety of different devices with different hardware architectures. The software on embedded devices is often interwoven with the hardware. In contrast to this, commodity computer systems typically use standard hardware with general-purpose operating systems and have significantly more computing power. And although these devices have a large number of different hardware configurations, they feature many abstraction layers and standard interfaces, allowing the replacement of individual components of commodity computer systems.

Furthermore, on embedded devices, there are many existing legacy applications of which the source code often is unavailable, especially considering the long lifetimes of these devices. Embedded devices regularly interact with the physical world and have requirements towards their timing behavior, so-called real-time constraints.

The main goal of this dissertation is to introduce novel security enhancements for existing legacy embedded devices, with a special focus on remote attestation approaches. Remote attestation allows offloading security checks to external entities. When developing new security solutions for legacy devices, it is important to solely use existing hardware and not require any new hardware extensions or changes to the device. Instead, we focus on using existing hardware features to implement security enhancements. We particularly consider the requirements mentioned earlier as well as the characteristics of embedded devices and show how to address these. The diversity of devices and their particular properties require unique solutions for different device types. We specifically focus on legacy embedded devices with real-time constraints. These devices do not have specific hardware security features that can be leveraged for remote attestation. Real-time systems have specific requirements towards their response times. During development, an extensive profiling phase ensures that the device can perform all critical tasks within the required timing constraints and therefore does not miss any deadlines. Hence, any new security enhancements for this type of device may not impact their runtime behavior.

1.2 Contributions and Outline

This thesis consists of five major parts. In the following, we give an overview of these parts, together with their contributions and related publications.

In Chapter 2 we provide a comprehensive background on embedded systems, particularly real-time systems, and remote attestation, with a special focus on software-based approaches. This background gives the necessary information for the following chapters.

Software-Based Attestation for Real-Time Systems. Smart factories, critical infrastructures, and medical devices largely rely on embedded systems that need to satisfy real-time constraints to complete crucial tasks. In Chapter 3, we present RealSWATT, the first software-based remote attestation system for real-time embedded devices. Software-based remote attestation works under the assumption that the attested device has a specific response time to an attestation request. The security of this attestation scheme relies on precise timing measurements of the attestation requests. This is in conflict with real-time constraints: Devices with real-time constraints need to operate under strict timing requirements. Any delay can have severe consequences. In contrast to previous remote attestation approaches for real-time systems, RealSWATT requires neither custom hardware extensions nor trusted computing components. It is designed to work within real-world IoT networks, connected through Wi-Fi. RealSWATT leverages a dedicated processor core for remote attestation and provides the required timing guarantees without hardware extensions. We implemented RealSWATT on the popular ESP32 microcontroller and evaluated it on a real-world medical device with real-time constraints. To demonstrate its applicability, we integrated RealSWATT into a framework for off-the-shelf IoT devices. Using this framework, we applied RealSWATT to a smart plug, a smoke detector, and a smart light bulb.

This chapter is based on the following publication:
Sebastian Surminski, Christian Niesler, Ferdinand Brasser, Lucas Davi, and Ahmad-Reza Sadeghi. "RealSWATT: Remote Software-based Attestation for Embedded Devices Under Real-Time Constraints". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS).* ACM, 2021.

User-Understandable Remote Attestation. In software-based attestation, there is the inherent problem of authenticity. The attested device cannot store secret keys to enable the verifier to identify the attested device reliably: If the attested device is compromised, the attacker has full access to the device's memory and may obtain any information, including all cryptographic keys. So, the attacker can replace the device with another device or even a simulation without being detected. To solve this issue, we developed SCAtt-man, which we present in Chapter 4. This remote attestation solution allows the user to identify the attested device with user-observable side-channels, thereby allowing a user-understandable attestation, increasing the user's trust in IoT devices and solving the device identification problem in software-based attestation. We designed SCAtt-man specifically with the user in mind. SCAtt-man deploys software-based attestation to check the integrity of remote devices, allowing users to verify the integrity of IoT devices with their smartphones. We implemented SCAtt-man into a smart speaker using a data-over-sound protocol. Our evaluation demonstrated the effectiveness of SCAtt-man against various attacks and its usability based on a comprehensive user study with 20 participants.

This chapter is based on the following publication:
Sebastian Surminski, Christian Niesler, Sebastian Linsner, Lucas Davi, Christian Reuter. "SCAtt-man: Side-Channel-Based Remote Attestation for Embedded Devices that Users Understand". In: *Proceedings of the 13th ACM Conference on Data and Application Security and Privacy (CODASPY).* ACM, 2023.

DMA-Based Remote Attestation. In traditional approaches, remote attestation uses a challenge-response protocol. In the simplest implementation, this is a hash of the attested memory regions. This hash is then sent to the verifier, which matches it to a list of well-known benign states. This simple process minimizes communication and can be adapted, for example, to cover additional security properties. However, the verification requires prior state space exploration. In Chapter 5, we propose DMA'n'Play, a new approach that directly connects the verifier to the attested device, thereby allowing the verifier to directly monitor the memory content of the attested device. We use the standard DMA feature to connect the verifier to the prover. We allow the developer to directly define correct states in a configuration file format. This approach is in contrast to other remote attestation schemes, where the verifier needs a complete list of all benign states. The verifier uses this configuration file and the compiled binary file of the attested device to identify regions in the memory of the attested devices. This technique makes the development of these configuration files a straightforward process. Furthermore, we introduce a small, low-cost external verifier device that can take over the role of the verifier, thereby allowing the verifier to be integrated into any embedded device. Due to the design of DMA'n'Play, the attested device cannot determine whether a verifier is connected or not, effectively solving the time-of-check/time-of-use problem.

This chapter is based on the following publication:
Sebastian Surminski, Christian Niesler, Lucas Davi, and Ahmad-Reza Sadeghi. "DMA'n'Play: Practical Remote Attestation Based on Direct Memory Access". In: *Proceedings of the 21st International Conference on Applied Cryptography and Network Security (ACNS).* Springer, 2023.

Hotpatching of Embedded Real-Time Systems. The main limitation of any remote attestation scheme is that it cannot actually prevent a compromise. By design, remote attestation can only detect a compromise. Typically, the reaction to a detected compromise is either a full stop of the system or entirely out of scope of the remote attestation scheme. A stop of the attested system is an adequate solution if the devices cannot be replaced or the integrity cannot be restored, but often inadequate in practice. In particular, remote attestation cannot solve the reason for the compromise. Hence, even when a benign state has been restored, the system can be compromised again. So in order to solve the root cause of the compromise, the device or application must be patched to remove the vulnerability that leads to the compromise. Patching is widespread in any computer system, including IoT and embedded devices. In many companies and institutions, patching is even mandatory. However, patching often induces side effects like interruptions, for example, by requiring restarts. These interruptions are often not an option in the case of highly critical real-time devices. Therefore, we developed HERA, a hotpatching framework for embedded real-time applications. It is able to apply patches in the background and then uses a standard hardware debugging feature on the popular ARMv7 microcontroller to activate patches without any timing delay. This approach makes HERA even suitable for the most critical real-time devices, which have hard real-time requirements and require exact predictability of their behavior.

This chapter is based on the following publication:
Christian Niesler, Sebastian Surminski, and Lucas Davi. "HERA: Hotpatching of Embedded Real-time Applications". In: *Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2021.

In Chapter 7 we sum up our findings. Additionally, we compare remote attestation with traditional security enhancements. We conclude this dissertation by showing worthwhile future research directions and open questions.

1.3 Additional Publications

Apart from the main publications which this dissertation is based on, the author was involved in several research projects, resulting in the following peer-reviewed publications:

- Moldovan, Christian, Florian Metzger, Sebastian Surminski, Tobias Hoßfeld, and Valentin Burger. "Viability of Wi-Fi caches in an era of HTTPS prevalence". In: *Proceedings of the 2017 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE, 2017.

- Surminski, Sebastian, Christian Moldovan, and Tobias Hoßfeld. “Saving bandwidth by limiting the buffer size in HTTP adaptive streaming”. In: *Krieger, Udo R.; Schmidt, Thomas C.; Timm-Giel, Andreas (Ed.): MMBnet 2017 – Proceedings of the 9th GI/ITG Workshop „Leistungs-, Verlässlichkeits- und Zuverlässigkeitsbewertung von Kommunikationsnetzen und Verteilten Systemen“*. University of Bamberg Press, 2017. DOI: `10.20378/irbo-49762`.
  Awarded with the best paper award.

- Surminski, Sebastian, Christian Moldovan, and Tobias Hoßfeld. “Practical QoE Evaluation of Adaptive Video Streaming”. In: *Proceedings of the International Conference on Measurement, Modelling and Evaluation of Computing Systems (MMB)*. Springer, 2018. DOI: `10.1007/978-3-319-74947-1`.

- Surminski, Sebastian, Michael Rodler, and Lucas Davi. “Poster: Automated Evaluation of Fuzzers”. In: *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2019.
  Awarded with the best technical poster award.

- 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: *Proceedings of the 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`.

- Tobias Cloosters, Sebastian Surminski, Gerrit Sangel, and Lucas Davi. “SALSA: SGX Attestation for Live Streaming Applications”. In: *Proceedings of the 7th IEEE Secure Development Conference (SecDev)*. IEEE, 2022. DOI: `10.1109/SecDev53368.2022.00019`.

- Sebastian Linsner, Kilian Demuth, Sebastian Surminski, Christian Reuter, and Lucas Davi. “Building Trust in Remote Attestation through Transparency”. *Under submission.*

CHAPTER 2

Background

Remote attestation is a security service that allows the verification of the integrity of a remote, untrusted system. Remote attestation is widely used in practice, for instance, to verify the integrity of SGX enclaves on Intel processors [61, 75] or the validity of Google services on Android devices using SafetyNet [127] or the Google Play Integrity API [126]. Samsung Knox allows remote integrity checks of a smartphone and determining if it is rooted or running unofficial firmware [232] as well as periodical checks of the Linux kernel, loadable kernel modules, and specific data structures in the kernel to prevent corruption of the device during runtime [232]. However, remote attestation is particularly popular for embedded systems that have limited computing resources and lack security features [3, 49, 84, 201] and sensor networks [254].

In this chapter, we provide the necessary background for this dissertation. First, we give the technical background on the specifics of embedded systems, particularly embedded devices with real-time tasks, the most critical class of such devices. Furthermore, we explain the typical vulnerabilities of these devices. This provides the necessary understanding of the systems on which we implement attestation schemes. Then, we give a broad overview of approaches to remote attestation and related threats, and describe the main challenges when developing remote attestation schemes. Finally, we provide a review of different remote attestation schemes for embedded devices, comparing both the attested properties and their implementation architecture.

2.1 Embedded Systems

Embedded systems are information processing systems embedded into enclosing products [178]. They are often deeply integrated into other devices. This results in an interwoven architecture of the hardware and software. Typically, these devices have a long lifetime as part of their enclosing device. As part of cyber-physical systems, embedded systems often operate in safety-critical environments. There, a malfunction or compromise can have devastating consequences. In addition to their interaction with the real world, embedded systems often even have real-time constraints. This means their correct operation also depends on the correct timing behavior of the device. Integrated into IoT devices, embedded devices are constantly connected to the Internet, making them a worthwhile target, for example for botnets [22].

2.1.1 Classification of Embedded Systems

In contrast to commodity computer systems, which have a predominantly homogeneous architecture, embedded systems frequently feature a compact and heterogeneous one [178]. There exist multiple classifications for embedded systems, depending on different system properties. For example, RFC-7228 categorizes embedded systems into three classes by memory size, while others focus on the software architecture. Muench et al. define three classes of embedded devices, depending on their software architecture and their operating system (OS) [188].

Type 1: Devices with General-Purpose OS. General-purpose operating systems are a popular choice for embedded devices due to their extensive functionality, interoperability, and long support times. Because of resource constraints often special lightweight versions of these general-purpose OS are used. There exists a wide variety of general-purpose OS for embedded devices, ranging from specialized Linux distributions such as OpenWrt[1], Ubuntu Core[2], customized Linux versions, created for example using Yocto[3] or Buildroot[4], to special Windows versions such as Windows IoT[5].

Type 2: Custom-Built OS. Type 2 devices feature a custom-built operating system, for example, a real-time operating system (RTOS). Real-time operating systems ensure the correct operation of real-time critical tasks by scheduling them accordingly. Despite the fact that sophisticated processor functions like a Memory Management Unit (MMU) may not be available in these systems, such operating systems still provide a logical separation between kernel and application code. Custom operating systems are particularly ideal for devices with low power consumption and little computing capability. Such operating systems can typically be found in specialized consumer electronics such as IP cameras or DVD players and include operating systems such as µClinux[6], Zephyr[7], or VxWorks[8] to name a few [188].

[1]`https://openwrt.org/`
[2]`https://ubuntu.com/core`
[3]`https://www.yoctoproject.org/`
[4]`https://buildroot.org/`
[5]`https://azure.microsoft.com/en-us/products/windows-iot/`
[6]`http://www.uclinux.org/`
[7]`https://www.zephyrproject.org/`
[8]`http://www.windriver.com/products/vxworks/`

Name	Data Size (e.g., RAM)	Code Size (e.g., Flash)
Class 0, C0	$\ll$ 10 kbyte	$\ll$ 100 kbyte
Class 1, C1	$\sim$ 10 kbyte	$\sim$ 100 kbyte
Class 2, C2	$\sim$ 50 kbyte	$\sim$ 250 kbyte

Table 2.1: Classification of embedded systems according to RFC-7228 [47].

Type 3: Embedded Devices with Monolithic Software. This class of devices uses a monolithic software architecture. In such systems, system and application code are compiled together. Often, the system even solely bases on a single control loop and interrupts to react to external peripherals. In contrast to types 1 and 2, these systems typically do not feature a hardware abstraction level. Such an architecture can be found in many small-scale devices. Often, hardware component controllers such as Wi-Fi cards, smart cards, or GPS receivers feature such a monolithic firmware [188].

RFC-7228 categorizes embedded systems into three classes by their amount of memory, as shown in Table 2.1 [47]. The amount of memory directly influences the amount and complexity of program code on the device. More RAM allows to store of more temporary data and is necessary to run multiple concurrent threats. This RFC focuses on the networking capabilities of the devices. While class 2 has sufficient resources for standard communication protocols, such as IP and HTTP, do class 0 and 1 devices require special reduced networking techniques such as CoAP (Constrained Application Protocol) [244].

Remote attestation schemes must also consider these limitations on software complexity and communication as well as other restrictions of embedded devices such as architecture, peripherals, and real-time constraints.

2.1.2 Typical Vulnerabilities in Embedded Systems

Now we take a look at typical vulnerabilities of embedded systems. As the goal of attestation is to detect compromises, it is important to understand the underlying root causes of the compromise. Furthermore, in order to test attestation schemes with real-world vulnerabilities, an overview of the most important types of vulnerabilities is necessary.

Commodity embedded systems often suffer from security vulnerabilities already known from classical computing, such as typical memory corruption attacks. However, standard computer systems feature many mitigations to hinder the exploitation of memory attacks such as ASLR [40], stack canaries [77], and non-executable memory [188]. On the contrary, embedded systems often do not have such mitigations [284]. Due to the limited hardware capabilities of embedded systems, they frequently lack standard security features of more sophisticated computer systems, such as memory management units [2]. Furthermore, the lower complexity of the software running on embedded systems combined with the lack of protection mechanisms eases the exploitation of memory corruption bugs.

While on a low level, embedded and IoT systems suffer from the same vulnerabilities, for example, caused by memory errors. On a high level these systems offer a different attack surface compared to commodity computer systems. A good overview of the most common vulnerabilities in IoT devices gives the Open Web Application Security Project (OWASP)[9]. OWASP is a non-profit organization with the goal to improve the security of web applications by providing openly available documents and tools, as well as organizing local chapters and conferences [209]. OWASP is well-known for its TOP 10 lists, which provide a consensus of the most severe security risks. It combines risks, threats, and vulnerabilities into a single concise list to raise awareness for the most common security problems. The TOP 10 list ranks security risks according to their severity, the frequency in which they occur, and the size of their potential effects. These lists are regularly updated and have become an important methodology to evaluate the security of web applications [210].

OWASP also provides a TOP 10 list of things to avoid when building, deploying, or managing IoT systems [208]. The first point (TOP 1) covers weak, guessable, or hard-coded passwords. This includes unchanged default passwords and weak passwords provided by users, as well as unchangeable credentials, backdoors, and development accounts included by the vendor, which is a common problem with IoT devices [273]. Often, network services are insufficiently secured (TOP 2). Especially when reachable via the Internet, these insecure services can be used to compromise or control devices, for example, manipulate software and configuration, control the device, or manipulate or exfiltrate data. Also, insecure ecosystem interfaces (TOP 3) are a common problem. This includes insecure web interfaces, back-end- or cloud interfaces, as well as interfaces for mobile apps. When insufficiently secured, these interfaces can be used to compromise the device. Typical problems are lacking or weak encryption, missing authentication or authorization, or a lack of input sanitation. Lack of a secure update functionality is also a common issue on IoT devices (TOP 4). Devices need an update functionality to fix discovered security issues, for example, by updating components. In order to apply updates securely, the update's integrity must be validated and a rollback must be prevented so that an attacker cannot install vulnerable outdated software versions. Furthermore, users must be notified about updates and the updates must be installed regularly. Related to this issue is the next problem, the usage of insecure or outdated components (TOP 5), which may have security vulnerabilities. Other points are insufficient privacy protection (TOP 6), insecure transfer and storage of data (TOP 7), lack of device management (TOP 8), insecure default settings (TOP 9), and lack of physical hardening (TOP 10).

[9]`https://owasp.org/`

Recent research confirms these findings. Alrawi et al. provide an extensive overview of security analyses of home IoT devices [12]. They identified vulnerable services, weak authentication, and insecure default configurations as common attack vectors. Also, cloud interfaces and insecurely configured services were typical problems. Often, IoT devices also used insecure communication techniques, for example by not using TLS encryption on the local network, or using vulnerable protocols such as Bluetooth or ZigBee. Furthermore, Alrawi et al. stated that security is heavily dependent on the vendors of the IoT devices. In most cases, the vendor had to release patches to fix these vulnerabilities. However, Alrawi et al. advised that vendors should provide a way for users to control, inspect, and evaluate their devices [12]. Remote attestation allows the implementation of such a functionality.

The security problems of embedded and IoT devices are further amplified by the typical properties of embedded devices. The limited hardware capabilities of these devices and simple architecture often lack common protection mechanisms known from commodity computer systems [2]. The interconnection of hardware and software results in customized software for each device and hardware configuration, hence is complex to change and update. In commodity computer systems with general-purpose operating systems, many abstraction layers separate operating systems, applications, and hardware. This allows the application of updates on individual components or replacement hardware parts or applications.

Therefore, when a vulnerability is found, the vendor needs to provide a patch or updated firmware to fix this vulnerability. There are many IoT devices with unpatched vulnerabilities, due to the lack of support by the vendor [56]. Additionally, embedded devices typically have a long life span. For example, airplanes have a design lifetime of more than 30 years [7], cars in the US are on average 12.1 years old [53], and industrial robots have a lifetime of ten years [8, 37]. This finding underlines the need for new security solutions that are also applicable to legacy embedded systems, thus devices that do not receive support by their vendor, for instance, with regular security updates.

2.2 Real-Time Systems

In contrast to commodity computer systems in which tasks are arbitrarily scheduled on a best-effort basis, real-time systems have strict requirements for the response times of tasks. That is, tasks need to be completed within certain deadlines. Often, real-time systems interact within the physical world, for example, as robot controllers or vehicle components. In such systems, inadequate response times, as well as malfunctions, have severe consequences. So, these systems have specific requirements towards correctness and safety as well as towards response times. During the development process, these requirements need to be considered, for example by testing or even formal validation for critical cases.

2.2.1 Classification of Real-Time Systems

Normal computer systems operate on a best-effort basis. In contrast to that, real-time systems have strict timing boundaries that have to be met for correct operation. There are different classes of real-time systems distinguished by their strictness, that is, the consequences of missing deadlines: hard, firm, or soft real-time requirements [247].

Hard Real-Time. In the class of systems with hard real-time requirements, no deadline may be missed, otherwise, there will be severe consequences, akin to a device failure. This is the most strict class of devices with real-time constraints. Typical examples of such systems with real-time constraints are control units for vehicles, for example, braking or engine control units where missing deadlines have a direct influence on the physical world. Also, industrial machines, cyber-physical systems, and medical devices often have hard real-time constraints [247].

Firm Real-Time. Firm real-time systems allow infrequent misses of deadlines. Missing a deadline does not have fatal consequences, they can be tolerated as long as they are infrequent and adequately spaced. However, when a deadline is missed, the result of the task is useless [247]. Examples of such systems are the transmission of audio or video, for instance in calls or video conferences. Due to the need for a short latency, delayed results are dropped. Industrial machines, where errors are compensated by quality control later on, are also examples of firm real-time systems.

Soft Real-Time. Systems that are neither hard nor firm are called soft real-time systems. In this class of systems, the value of a task starts to degrade after its deadline. But the system can tolerate even frequent misses of deadlines. However, missing deadlines reduces the system's overall performance. A good examples of soft real-time systems are IoT devices that take measurements and control other devices. Consider an IoT weather station that monitors the environment and sends these measurements at regular intervals. Measurements may be late as long as they are still relevant. A smart heating system is also a soft real-time system. The system tolerates even frequent deadline misses. However, the value of the delayed measurements decreases as the delayed measurements make the control loop of the system less responsive. However, as long as there are sufficient measurements, the functionality of the system is not impaired. Also, transmission in audio and video streaming applications can be considered as soft real-time systems. Even frequent missed deadlines only degrade the overall user experience, because of increased latency.

As real-time devices, especially hard real-time systems, are often safety-critical, there are specific procedures and regulations to be considered during development. In the so-called profiling phase, it is ensured that under all circumstances the real-time deadlines are met. Changes to either hard- or software induce a new validation process as well as a re-certification in the case of safety-critical devices, for example in avionics [230]. In real-time systems, an operating system is often used to schedule tasks so that all deadlines are met.

2.2.2 Scheduling and Real-Time Operating Systems

The described real-time capabilities are often ensured by the use of a real-time operating system (RTOS). The RTOS manages and schedules tasks so that all deadlines and constraints are fulfilled. One of the most crucial parts of a real-time operating system is the scheduler. The scheduler determines which threads are executed on which processor core. Threads have different states: Runnable threads can be executed, and blocked when they are waiting, for example, for an event such as I/O [39]. There are different strategies to manage these threads while maintaining real-time deadlines. A popular scheduling algorithm is fixed priority scheduling. It works on a first-in/first-out basis enhanced with priorities. This is how it works: Each thread has a priority. The processor runs the runnable thread with the highest priority. When a higher-priority thread becomes runnable, the scheduler preempts the running thread and executes the thread with higher priority. In order to prevent starvation of threads, that is, when a task is never executed because there are multiple threads on the same priority level, this scheme is often combined with a round-robin scheduling approach. In round-robin scheduling the scheduler switches to another thread on the same priority level after a certain time [39]. Fixed priority scheduling has static priorities for threads. In contrast, the earliest deadline first scheduling algorithm has dynamic thread priorities. Whenever possible, this scheduler runs the thread with the earliest deadline. While in theory fixed priority scheduling is proven optimal for single-core systems and allows full CPU usage in all circumstances, in practice this scheduling algorithm can cause significant overhead due to dynamic updates of the thread priorities.

Embedded devices have a wide variety in architecture. While some devices are so-called bare-metal systems, many embedded systems run operating systems (OS). The AspenCore Embedded Market Study finds that 65% of all projects use an operating system, real-time operating system, kernel, software executive, or scheduler of any kind [34]. Popular operating systems for real-time systems are FreeRTOS and Zephyr. According to the Eclipse Developer Survey, Linux (31%), FreeRTOS (28%), and Windows (12%) are the most popular operating systems for constrained devices and edge nodes. Zephyr is in fourth place with 10% [88]. The AspenCore Embedded market study has similar findings. The most popular operating systems

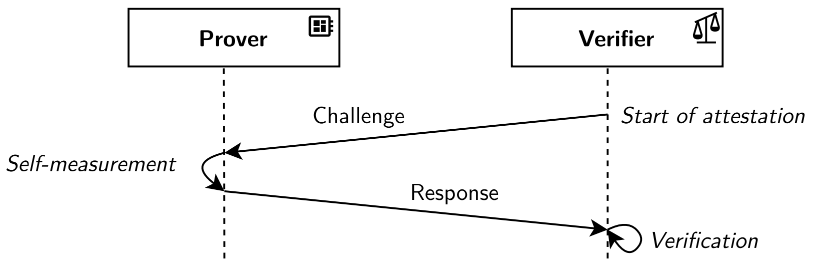


Figure 2.1: Remote attestation typically uses a challenge-response protocol. The verifier initiates an attestation run by sending an attestation request to the prover. The prover performs a self-measurement which is then sent back to the verifier.

are embedded Linux, custom operating systems, and FreeRTOS [34]. In this thesis, we will use FreeRTOS as it is a popular open-source real-time operating system.

2.3 Overview on Remote Attestation

Remote attestation is a security service that allows a system to check the integrity of a remote device. The main challenge in remote attestation is to obtain a trustworthy measurement of an untrusted device. Attackers can modify the software and internal state of the attested device to forge self-measurements. To perform secure self-measurements, the attested device inherits an *attestation function* that allows the attested device to prove its integrity. Therefore, the attested device is also called the *prover*. The system that verifies the integrity of the prover is hence called the *verifier*.

There are several main requirements towards a secure attestation scheme.

Integrity. The attestation function must reflect the real device's state so that the verifier can actually detect deviations of the attested device from benign states, such as installed malware, altered configurations, or other compromises. Therefore, the attestation report that the attested device sends to the verifier must be benign and may not be altered.

Authenticity. The measurement must actually come from the attested device. Attackers may not replace the device with another device or a simulation or route the attestation request to another device.

Freshness. The measurement must be recent, and the attacker may not delay or replay old responses.

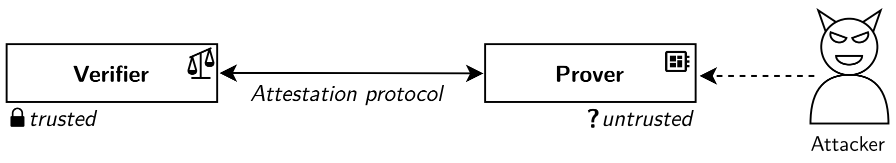


Figure 2.2: Attestation allows verifying the integrity of a remote, untrusted device.

Remote attestation is typically implemented using a challenge-response protocol. Although there also exist complex schemes for sensor networks [254] or autonomous devices [4, 143, 160], we focus on a classic setting with a single prover and verifier, as this setting is most common for embedded and IoT devices. Figure 2.1 shows such a remote attestation protocol. The verifier sends an attestation request 'challenge' to the attested device, the prover. The prover performs a self-measurement using its integrated attestation function and sends the result 'response' back to the verifier. The verifier then verifies this measurement and thereby can detect whether the attested device is in a benign state or whether it has been compromised. To prevent replay attacks, the verifier sends a nonce in the attestation request, which the prover then integrates into its response. In Section 2.5, we explain all challenges towards secure remote attestation schemes in detail.

There are two common ways to classify remote attestation schemes.

Attestation Type. The attestation type describes the properties that the attestation scheme covers. For example, static attestation checks static properties, such as the integrity of the program code and data. Runtime attestation also covers dynamic properties such as the control flow and data flow of the attested device or application.

Architecture. The architecture of the attestation scheme describes how the attestation function is implemented. As discussed, performing a secure self-measurement is a challenging task. Many attestation schemes require special hardware extensions or trusted computing components, while software-based solutions do not have specific hardware requirements.

In Section 2.6, we elaborate in detail on the different attestation types. Finally, we compare different attestation architectures in Section 2.7.

2.4 Threat Model of Remote Attestation

Figure 2.2 shows the typical threat model in remote attestation. Remote attestation assumes an untrusted remote device called the prover. The attacker can compromise the attested device, for example, by exploiting software defects such as memory errors, and install malware, altering configuration data, or performing runtime attacks such as return-oriented programming [223] or data-only attacks [140, 148]. In line with previous work, in this thesis, we exclude physical attacks on attested devices [3, 4, 84, 201, 203, 240, 241]. This is a standard assumption in single-prover attestation protocols [3]. In the case of safety-critical embedded devices, when an attacker gains physical access to a device, this often anyway has severe consequences. An attacker with physical access can, for example, destroy devices, unplug cables, or manipulate external peripherals. By design, remote attestation cannot prevent such manipulations. In any remote attestation scheme, the verifier is assumed to be trusted. However, attacks on the prover by a malicious verifier should also be considered. A malicious verifier—either a compromised verifier, an impersonation, or simple replay attacks—could maliciously trigger the prover to perform an attestation, for instance, by sending attestation requests, thereby impairing the performance and behavior of the attested device. This behavior could also lead to a complete denial of service, rendering the attested device unusable [50]. In this thesis, we provide a detailed threat model for each developed attestation technique in the corresponding chapters.

2.5 Challenges in Remote Attestation

Developing remote attestation schemes is a challenging task. Performing a secure self-measurement of a remote, untrusted device is a complex problem. Remote attestation schemes assume powerful attackers that can fully compromise the attested device. So the attested device cannot be trusted, including any software running on it. This includes manipulating or running any code or changing or exfiltrating any data on the attested device. Regardless of this, attackers may not be able to prevent an attestation measurement, falsify the attestation reports, or be able to compromise the device without being detected. Consequently, implementing secure remote attestation schemes is a complex task as they offer a large attack surface. For example, attackers can copy all the data on a compromised device to another instance and thereby replace the attested system with another device or a simulation. This data includes all cryptographic keys used for authentication, allowing the attacker to impersonate the attested device. Another popular attack technique is to remove or hide the traces of a compromise before the device is being attested. This problem is known as the time-of-check/time-of-use (TOCTOU) problem. In this type of attack, the device is compromised in between two attestation runs without being detected. In addition, attackers could also delay the attestation measurement to increase the time span between two attestation runs and hide attacks.

Communication is also an important aspect of remote attestation schemes. By design, remote attestation requires constant communication between the prover and the verifier. This is fine in scenarios with reliable network connectivity, small numbers of attested devices, and unlimited energy supply. However, when scaling remote attestation to a large number of devices, communication becomes a significant restriction. All devices require continuous communication with the verifier. But, especially in wireless settings, communication bandwidth is limited. Continuous communication is even more challenging due to limited network coverage when attesting mobile devices such as vehicles, drones, or planes. In addition, depending on the usage scenario, communication may not be possible during specific times. Furthermore, communication is also costly in terms of resources. Transmitting data consumes significant amounts of energy. This is certainly an important aspect, for example, when attesting battery-powered devices.

In particular, the following general challenges must be considered when developing new remote attestation schemes.

Challenge 1: Secure Self-Measurement. The attestation function must perform a secure self-measurement, even under full system compromise. The measurement must reflect the device's actual state. Attackers may not be able to alter the result of the attestation function.

Challenge 2: Authenticity. The attestation scheme must guarantee that the measurement indeed comes from the attested device. Even when fully compromising the attested device, the attacker may not be able to impersonate the attested device. This means attackers may not be able to relay attestation requests to other devices or replace the attested device with another device or a simulation.

Challenge 3: Time-of-Check/Time-of-Use. The time-of-check/time-of-use (TOCTOU) problem describes the inherent delay between a check and the actual usage. When an attacker between two checks can compromise a device and hide the traces of the attack, attestation will not detect it, thereby rendering the attestation useless.

Challenge 4: Communication. Remote attestation requires communication between the attested device and the verifier. These aspects must be considered, especially in scenarios with large fleets of attested devices, wireless communication, unreliable transmission, as well as limited bandwidth and connectivity.

Challenge 5: Hardware Requirements and Legacy Devices. There are billions of legacy embedded and IoT devices which cannot be replaced. Security solutions for these systems must not require additional hardware extensions, extra sensors, or new communication technologies since deploying new hardware is costly and often impractical, hindering wide adoption.

Further Challenges. Besides these general challenges, there are different types of embedded devices that have specific additional demands. When implementing remote attestation on devices that operate under real-time constraints, these timing constraints may not be impaired. In particular, the attestation may not interfere with the real-time operation. So either the attestation task or the real-time operation must be paused or aborted, or both tasks must be performed completely in parallel without any side-effect on each other's operation. At the same time, the TOCTOU problem described in Challenge 3 must be considered. Attackers may not be able to hide their traces intentionally by preventing a device from being attested, for instance, by delaying measurements until the attacker is able to hide all attack traces.

There are also other practical requirements to consider in remote attestation. A secure remote attestation scheme itself is not sufficient. As described in Challenge 4, the attested device also needs to communicate reliably with the verifier. When the verifier does not receive the attestation reports, a compromise cannot be detected, rendering the remote attestation useless. Furthermore, the complexity of the verification task is important. Especially when scaling remote attestation on large installations, such as networks of IoT devices or large fleets of vehicles or drones, the computational demands of the verification become relevant. A simple verification with little communication demands allows the deployment of remote attestation on a large scale.

Also, the response to a detected compromise is essential. First, this covers a reliable detection with preferably no false positives. Second, how to react to a compromise must be considered. Remote attestation itself does not prevent any attacks. Depending on the device type, different reactions are appropriate. For example, in the case of a compromised medical device, the personnel can be alerted. In the case of an industrial robot, the machine can be stopped or brought into a fail-safe state. In the case of consumer IoT devices, users need to be notified and instructed to resolve the problem. This requires a proper process to restore the device's integrity and adequate communication with the user to ensure good usability and user acceptance. Especially when attestation schemes for IoT devices require user interaction also, the usability and user's understanding must be considered. Research shows that users tend to discard or circumvent mechanisms they do not understand or trust [233].

Summing up, the large variety of embedded devices has different requirements for secure remote attestation schemes. It is difficult to address all of these requirements in one remote attestation scheme. There does not exist a one-size-fits-all solution that covers all needs. In this dissertation, we propose different remote attestation schemes. For each scheme, we will show how these general challenges are addressed.

2.6 Attestation Types

There are two approaches to classifying remote attestation schemes. First, based on the properties that the attestation scheme measures. Second, using the architecture of the remote attestation scheme. We first focus on the different system properties that remote attestation schemes can measure.

2.6.1 Static Attestation

The most basic type of attestation is static attestation. In these schemes, the attestation verifies the integrity of a system or a device by analyzing its firmware, software, or static configuration. This attestation mechanism is able to detect firmware modifications, such as persistent malware or altered configurations. Because static attestation measures data that is subject to little changes, both the measurements and the verification do not require complex processes or computations. Typically, the prover hashes the attested memory regions and then sends the resulting hash value to the verifier. The verifier then compares the received measurement to a list of benign states. This makes static attestation well-suited for embedded devices with limited processing power and large fleets of devices, as due to the simple verification process, a single verifier can attest a large number of devices. Static attestation is widely used. There are static attestation schemes specifically designed for all kinds of devices, including IoT devices, devices with real-time constraints, sensor networks, and industrial control systems. However, static attestation cannot detect runtime attacks, for example, using return-oriented programming (ROP) [223, 242], that target the program's execution instead of the static binary. Runtime attacks exploit applications by modifying state information during execution while leaving the program code unchanged. To detect such attacks, this state information needs to be covered by the attestation. Dynamic attestation schemes such as control flow and dats flow attestation address the shortcomings of static attestation as they include runtime properties and involve verifying the state of the device while it is running.

2.6.2 Control Flow Attestation

One type of dynamic attestation is control flow attestation. Static attestation only attests the integrity of the binary data of the attested application. In contrast, control flow attestation includes the actual execution of the application in the attestation. This is achieved by measuring the control flow of the attested application. This enables the verifier to detect deviations in the control flow that are, for instance, caused by runtime attacks such as jump-oriented programming [42, 59] or return-oriented programming [223, 242]. In code-reuse attacks, the attacker reuses existing code snippets, called gadgets, to manipulate the program's control flow and execute the malicious code. Chaining these gadgets allows to perform arbitrary actions. Research showed that such attacks are Turing-complete [242].

In contrast to static attestation, where attestation can take place at dedicated times, control flow attestation needs to monitor the execution continuously. This requires a more sophisticated architecture, as every branch, that is, an alternative in the control flow, needs to be recorded. The verification of the execution of a program requires a full graph of the benign control flow. A control flow graph represents the control flow in a program. Each node consists of a set of assembly instructions, and the edges represent branch instructions, which influence the program flow. The size of the control flow graph is dependent on the complexity of the attested application. For control flow attestation, nodes need to be identified, for example, using unique labels [3]. A complex application results in a large control flow graph, a so-called state space explosion [174]. Control flow graphs can become extensive for larger applications, resulting in a larger effort to verify the prover's measurements and limiting the complexity of applications that can be attested. Special challenges in control flow attestation are loops. Loops lead to a large state space, and applications may have infinite loops. Especially IoT and embedded devices that perform control or measurement tasks are often implemented using endless loops. Therefore, loops have to be handled separately to prevent control flow graphs from exploding.

In the following, we provide an overview of different control flow attestation schemes. C-FLAT performs control flow attestation using ARM TrustZone [3]. However, in order to monitor the control flow, C-FLAT requires a full instrumentation of the attested application. On every branch, a switch to the secure world of TrustZone was required, resulting in high overhead. ATRIUM can detect control flow deviations even if the memory is physically changed. This is achieved by including the executed instructions in the attestation. LO-FAT tackles the limitation of the high overhead induced by the attestation as it implements the control flow attestation completely in hardware [85]. However, this requires complex custom hardware extensions. TinyCFA is a hybrid attestation scheme for control flow attestation that requires minimal custom hardware and is based on the VRASED framework [205].

2.6.3 Data Flow Attestation

Data-only attacks focus on the manipulation of data without changing the control flow. As the control flow is not changed, these attacks are not covered by control flow attestation. To detect this type of attack, the attestation scheme also needs to cover the application's data. Developing data flow attestation schemes is highly challenging as all data of the attested application needs to be included. Furthermore, small deviations in the data may have severe consequences. For instance, consider an application that performs permission checks. Changing one bit of data on the user ID can have a significant impact. When unprivileged users are able to set their user ID to 0, they can obtain root privileges [64]. There have been data-only attacks on both server software [140], such as web servers [148] and FTP servers [64] as well as client applications, for example, web browsers [224]. Furthermore, there are more complex data-only attacks. These attacks use so-called data-oriented programming to run arbitrary code without violating control flow constraints. Research found that this type of attack is actually Turing-complete [148].

There are different attestation schemes that cover the program's data during runtime. For example, OAT introduces the concept of operational integrity for critical sections in embedded devices [265]. DIAT allows attesting the integrity of the data of the selected information in autonomous devices during both the processing on the devices and during the transmission between devices [4]. DIALED is a hybrid data flow attestation scheme that builds upon the proof of execution of APEX and control flow attestation of TinyCFA [204]. LiteHAX attests the control flow and the data flow using a custom processor extension. It introduces instruction tracers for control flow instructions and load/store instructions. As this approach relies heavily on hardware, the performance of the attested application is not impaired. Furthermore, LiteHAX does not require any instrumentation of the attested application [84].

2.6.4 Other Properties

In addition, there exist other remote attestation schemes that measure other properties. Thereby it is possible to implement more specialized security guarantees using remote attestation while reducing the complexity of the implementation compared to sophisticated runtime attestation schemes, such as control flow or data flow attestation.

Proof of Execution. One concept is 'proof of execution'. This means attesting that a specific program function has been executed, which also requires measuring the control flow. However, the proof of execution only covers a small part of the program, drastically reducing the effort for the attestation measurement and verification. This is implemented, for instance, in the APEX framework and allows to check that certain code sections have actually been executed [203]. APEX builds upon the VRASED framework [201]. Delegated attestation is another attestation scheme for proof of execution. It allows software-based proof of execution by using a gateway and attestation proxy to monitor the execution behavior of the attested devices [17].

Proof of Update. Attestation can also be used to verify that an update has been applied. Such an attestation scheme allows to assure that a vulnerability has been patched. There are numerous attestation schemes that allow proof of update. For instance, SCUBA implements a proof of update mechanism for sensor nodes [239]. PoSE is a hybrid attestation scheme for secure update and erasure [215]. Feng et al. extend PoSE to use physically unclonable functions (PUFs) as root of trust for the remote attestation [111]. ASSURED combines a secure update mechanism for embedded devices with remote attestation [33].

Proof of Erasure. This attestation scheme allows one to verify that the data on a device has been deleted and the device has been reset to a defined state [19]. Another variance of proof of erasure is 'proof of reset', which checks whether the attested system has been reset to a defined state. For instance, Verify&Revive is a software-based attestation scheme that is able to reset the device upon detection of a compromise [16]. A combination of proof of update and proof of reset allows to assure that a device has been updated and reset to a secure state. For instance, Pure is a hybrid attestation scheme based on the VRASED attestation framework [201] and allows for a proof of update and proof of reset [202].

These different security properties can be used and combined to assure the required specific behavior of embedded devices. This way, it is possible to customize remote attestation to specific needs, thereby considering security requirements and hardware constraints.

2.7 Attestation Architectures

In general, there are different approaches to implementing remote attestation. They vary in the hardware required to perform the attestation. Software-based attestation does not require any special hardware, while hybrid and hardware-based schemes need special hardware extensions. As mentioned before, the main challenge in remote attestation is to obtain a trustworthy measurement from an untrusted device despite the strong attacker model. This applies to all types of attestation. Even under a full system compromise, in no case the attacker may be able to forge attestation results or hide attacks from the attestation.

2.7.1 Hardware-Based Attestation

A straightforward approach to remote attestation is to use a secure and trusted subsystem to take measurements of the device. Such a subsystem can then both perform the attestation and store a secret key to authenticate itself to the verifier to prevent impersonation attacks. Without such an authentication, the verifier cannot reliably identify the attested device.

Such trusted subsystems include trusted platform modules (TPM) and trusted execution environments (TEE). A TPM is a secure co-system with a dedicated microprocessor. This module is isolated from the system and specifically designed for security purposes, such as securely storing cryptographic keys or providing security functions. For instance, TPMs are used as a root of trust for secure boot, where the boot process is monitored [274]. While secure boot ensures the initial integrity of a system during startup, attestation measures the system's state during runtime. TPMs are developed and standardized by the Trusted Computing Group[10]. There exist many approaches to using TPMs for attestation. For example, property-based attestation can be implemented using TPMs [229]. TPMs have also been used to implement remote attestation in sensor networks [141, 267]. ReDAS is a framework that allows dynamic attestation using TPMs [155].

While TPMs allow secure key storage and boot security, they cannot provide secure computation outside of the TPM. In contrast, a trusted execution environment (TEE) can provide secure execution of code even on an untrusted or potentially compromised system. While a TPM is a dedicated, external hardware module, the TEE is a secure subsystem within a processor. Examples of platforms with TEEs are Intel SGX [75] and ARM TrustZone [24]. Trusted computing technology is integrated into many sophisticated off-the-shelf processors as they are used in servers, personal computers, or smartphones. In contrast, less powerful processors, as they are used in embedded and IoT devices, do not feature such TEEs due to cost reasons. TEEs can be used to reliably monitor the data and execution in the normal world from within the secure world. Hence, the monitoring cannot be influenced. For example, Abera et al. proposed C-FLAT, a framework to remotely verify the control flow of applications running in the normal world using the TrustZone of an ARM processor [3].

However, these trusted computing modules are often not available on legacy embedded devices due to cost reasons. TrustZone is also used in practice to implement remote attestation. For instance, Samsung Knox uses TrustZone to verify the integrity of a smartphone by checking if it is rooted or running unofficial firmware [232]. Furthermore, Samsung Knox periodically checks the Linux kernel, loadable kernel modules, and selected data structures in the kernel to prevent corruption of the device [232].

2.7.2 Software-Based Attestation

In contrast to hardware-based attestation, software-based attestation (SWATT) schemes do not need any trusted computing components. These schemes are purely implemented in software, thereby eliminating the major limitation of hardware-based attestation approaches. Therefore, software-based attestation (SWATT) is well-suited for commodity hardware and legacy devices. Software-based attestation works under the assumption that the attacker cannot change the target device and its computing capabilities. The attacker is not able to introduce further computing resources.

[10] `https://www.trustedcomputinggroup.org`

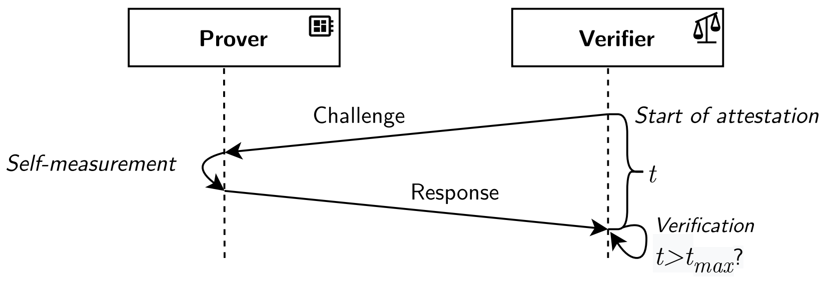


Figure 2.3: In software-based attestation, the verifier measures the time until the attestation report of the prover is received.

Hence the execution time of specific operations is bound. That is, the attacker has no physical access to the attested device. This circumstance allows to measure the response time of the prover: if it takes longer than expected, the system is likely compromised. The attacker cannot forge these results, as this would require further operations so that the response cannot be sent in time [241]. Based on these assumptions, the verifier precisely measures the response times for the attestation requests. This includes the time it takes for the prover to compute the attestation report as well as the communication overhead between the prover and the verifier. If the response times differ from the expected values, the prover device is assumed to be compromised. This induces many requirements on the implementation of the attestation logic and communication [32, 241].

Figure 2.3 shows the attestation process in software-based attestation. Similar to traditional attestation, the verifier sends a challenge to the prover. The prover then performs a self-measurement using its attestation function and returns the result. The verifier measures the response time of the prover and uses the result to determine the prover's integrity. Given the fact that the attestation function is optimal, it cannot be accelerated. As the trusted verifier chooses the challenge, and the prover requires the challenge to start the attestation function, this process cannot be started earlier. In addition, the challenge ensures freshness and prevents the pre-computation of the attestation result.

However, as the security of software-based attestation is solely based on the timing of the attestation function, this poses two general challenges. First, the response time must be precisely measured. As the verifier measures the response time, this includes the attestation function on the prover as well as the transmission time between the prover and verifier. Second, the implementation on the prover must be optimal and must cover all memory on the attested device. Furthermore, the attacker may not be able to free up any memory, such as by means of compression. This can be achieved by filling up all empty memory on the prover. Third, if the attacker is able to accelerate the attestation function, for instance, by further optimizations or shortcuts, the attacker can then use the free time to compromise the device or obfuscate traces of attacks, rendering the attestation useless. In fact, the correct and secure implementation of software-based attestation is complex and error-prone [57]. However, when no hardware support for other attestation schemes is available, software-based attestation is a viable solution. This is why software-based attestation is, for instance, commonly used in sensor networks [254].

2.7.3 Hybrid Attestation

Hybrid attestation schemes describe a hardware/software co-design that performs remote attestation in software supported by custom hardware extensions. These hardware extensions allow to securely store keys or track the execution of commands on the processor. This gives a root of trust so that the attested device cannot be emulated or replaced by the adversary. There exist various proposals for attestation to use customized hardware to provide remote attestation functionality [49, 91, 159, 201]. SMART is the first attestation scheme that builds upon simple, customized hardware [91]. While the attestation is performed in software, the secret key is protected using custom hardware functionality so that it is not leaked during attestation.

While SMART uses a small extension to protect the keys used for remote attestation, it lacks more sophisticated features, such as the ability to update the attestation code. Furthermore, SMART requires the attestation to run atomically, which is a major drawback in many application scenarios where the attested device may not be interrupted by the attestation process, for example, in real-time systems. On the contrary, TrustLite does not have these limitations, as it allows isolating different software modules with hardware modifications of the Memory Protection Unit (MPU) and the exception engine of the processor [159]. TyTan builds upon TrustLite and extends it so that it is able to run applications with real-time requirements [49]. VRASED is a formally verified hardware/software co-design for remote attestation [201] and allows verification of the state of the device memory. It has been extended to also verify reset, erasure, and update of devices [202] and also attest that code has actually been executed [203]. Furthermore, there exists an attestation scheme that allows data flow attestation based on this architecture [204].

While these hybrid attestation schemes have many advantages, the required custom hardware extensions are not available on already existing legacy devices. The creation of customized hardware is complex and adds significant costs to the manufacturing process, especially in comparison to widely used off-the-shelf hardware. Such customization is only practical for a large number of devices. Embedded systems often use off-the-shelf microprocessors with generic hardware modules. For the implementation of a hybrid attestation scheme, the device itself needs to be replaced. Thus, hybrid attestation is no viable option for legacy devices.

2.8 Comparison of Attestation Schemes

There exists a large variety of different attestation schemes. They differ in their architecture as well as their security guarantees. When selecting an attestation scheme, it is important to consider the particular use case. Static attestation ensures the integrity of a device's firmware. The verification is efficient to implement, as the verifier only needs to check the hash value of the firmware. Attestation can be performed at arbitrary times, minimizing the impact on the system's performance as well as reducing communication between the prover and the verifier. However, static attestation cannot detect runtime attacks. To cover runtime attacks, dynamic properties must be considered, such as the control flow or data flow of the application. Furthermore, there exist attestation schemes that cover special use cases, such as proof of update or secure reset. These schemes allow for giving specific security guarantees. For instance, a proof of reset can restore a secure state of a possible compromised device without implementing more complex attestation schemes such as a full control flow attestation.

Another important aspect is the attestation architecture. When trusted hardware components are available, hardware-based attestation is an elegant solution, as the attestation functionality can be separated from the attested device. However, if trusted hardware is not available, then hybrid attestation schemes are worthwhile options. Hybrid schemes require minimal hardware modifications but no complex trusted hardware. If the necessary hardware modifications are not possible, for example, when developing attestation for existing legacy devices, the only option are software-based attestation schemes, despite their complex implementation.

Summing up, there does not exist a one-size-fits-all solution to remote attestation. Choosing the suitable attestation scheme requires careful analysis of the particular use case. It is important to consider the device type, the available hardware, and the implementation complexity of the attestation scheme as well as the security guarantees the attestation scheme needs to give.

CHAPTER 3

Attestation for Real-Time Applications

Commodity real-time embedded systems often suffer from security vulnerabilities already known from classical computing. However, due to resource constraints, embedded devices often lack basic security mechanisms that are common in most other types of systems [12]. At the same time, real-time applications, which are essential in many safety-critical domains, place highly conservative requirements to guarantee the strict real-time operation [252]. The need to secure embedded devices is further amplified by the trend of the Internet of Things (IoT) to connect previously unconnected and isolated devices to the Internet to enhance features and services. In particular, any malfunction in real-time devices can have fatal consequences since they perform highly critical real-world tasks in many safety-critical domains such as cyber-physical systems, medicine, and transportation. This leads to large vulnerable ecosystems consisting of millions of devices.

Legacy real-time embedded devices typically lack hardware features, for example, secure boot or trusted execution environments (TEEs). Moreover, they are commonly integrated into machines and run customized software and hence, cannot be simply replaced. As eluded earlier, the integration of software-based security mechanisms such as control flow integrity (CFI) [1] have a direct influence on the execution times of tasks [79, 277]. This is a crucial issue since real-time systems must conform to tight timing behavior and undergo a lengthy development and profiling phase. Any modifications to the execution, including those made through instrumentation, such as the integration of control flow integrity [79, 277], or abnormality monitoring [246], have an impact on the device's runtime behavior.

Hence, adequate security solutions for real-time applications must have a strictly limited impact on the real-time operations [252]. Currently, there exist no practical solutions that can tackle these challenges. Furthermore, the handling of detected suspicious or malicious behavior is an important question for critical real-time systems. Solutions like control flow integrity may terminate if an illegal path is executed.

At first glance, a promising solution to tackle these challenges seems to be remote attestation (RA), as it offloads the verification of the monitored device to an external trusted party. RA allows a trusted party, called verifier, to gain assurance about the correctness of the state of a remote device, called prover. It has been used for embedded devices [3, 49, 84, 201] and sensor networks [254].

At second glance, the usage of remote attestation for such devices is a complex challenge. The main difficulty for attesting real-time devices is, however, to utilize the attestation independently from the execution of the monitored application. Moreover, another vital aspect of remote attestation is to get a genuine attestation report from an untrusted device. An attacker could forge the attestation report, for example, by using a different device or an emulation of the attested system. There have been a variety of proposals for attestation schemes to address this issue: (1) hardware-based using trusted computing [3], (2) hybrid using custom hardware extensions [49, 84, 91, 159, 201], and (3) software-based [241, 254]. We give an overview on attestation architectures in Section 2.7. However, none of these solutions is an option for legacy real-time embedded devices: While hardware-based and hybrid approaches to attestation require changes to or customization of the underlying hardware, software-based attestation poses strict timing assumptions on the response of the prover, and the verifier induces many requirements upon the implementation of attestation logic and communication [32, 241]. These timing requirements influence the runtime behavior of the attested device and therefore are not suited for devices with real-time requirements.

Contributions. In this chapter, we present RealSWATT, the first software-based remote attestation system for real-time embedded devices. In contrast to previous remote attestation approaches for real-time systems, RealSWATT does neither require custom hardware extensions nor trusted computing components. It is designed to work within real-world IoT networks, for instance, connected through Wi-Fi. Thus, RealSWATT can secure legacy devices—even those that operate under real-time constraints—without the need to alter the hardware.

Software-based attestation poses strict timing assumptions on the response of the prover, and the verifier induces many requirements upon the implementation of attestation logic and communication to enable a secure self-measurement of the attested device [32, 241]. These timing assumptions are in an inherent conflict with real-time constraints, as both systems require immediate reactivity at any time. RealSWATT is the first remote attestation framework to solve this conflict. We leverage off-the-shelf hardware and do not require any trust anchor on the attested device, unlike hardware-based and hybrid attestation approaches that require changes to or customization of the underlying hardware.

A key aspect of our design is based on the observation that many modern low-cost embedded systems, such as the ESP32, are built on a multi-core architecture where the cores are often not fully utilized. Especially in a real-time context, multi-threading is hard because meeting all deadlines under all circumstances is of utmost importance. So in practice, critical tasks are often not scheduled in parallel. Moreover, in specific areas, for example, avionics, there are even regulations to limit the usage of additional cores [60]. As a result, one or more processing cores are idle. We leverage this circumstance and utilize an idle processor core to develop a new attestation framework. This allows the attestation and the real-time tasks to be properly scheduled by the underlying real-time operating system and makes RealSWATT suitable for legacy embedded devices in the industry, medicine, and cyber-physical systems.

However, the usage of multi-core processors involves tackling several new challenges: while the benign execution only uses one processor core, an attacker can now use all processor cores in order to forge an attestation report. We address this issue by selecting adequate cryptographic functions that parallelization cannot accelerate. Furthermore, communication, especially in wireless networks commonly deployed for IoT devices and cyber-physical system setups, is prone to variation in transmission times. This also conflicts with the strict assumptions for software-based attestation [32].

Because of these required strict assumptions and shortcomings like the vulnerability against compressing attacks [57] software-based attestation has been assumed insecure and infeasible in practice, receiving only little to no attention. It requires specific assumptions about the execution speed of the prover logic on the attested device as well as precise timing measurements, making the implementation challenging [241]. However, software-based attestation is a good fit for legacy devices, where other attestation schemes are simply unavailable due to the lack of specialized hardware on the given device. Thus, we have re-evaluated software-based attestation and solved several challenges, which allow us to deploy a software-based attestation scheme in a real-world scenario.

We developed a new concept called continuous attestation, where the verifier sends the next attestation request before receiving the prover's previous response. This way, the prover can start the next attestation run directly after completing the previous one without any waiting time. This procedure omits the transmission and verification time so that variations in transmission time do not influence the attestation. Furthermore, our continuous attestation approach induces the strict requirement that the attacker cannot run two attestation protocols in parallel in order to get a time span in which no attestation is performed, thus effectively solving the time-of-check/time-of-use (TOCTOU) problem. Combining these techniques (using a multi-core architecture and continuous attestation) ensures that RealSWATT can reliably attest real-time embedded systems in real-world wireless networks without impairing real-time operation. To show the applicability, we implemented RealSWATT on the popular ESP32 microcontroller, and we evaluated it on a real-world medical device with real-time constraints. To demonstrate its practicability, we furthermore integrated RealSWATT into ESPEasy, a framework for off-the-shelf IoT devices, and applied it to a smart plug, a smoke detector, and a smart light bulb.

In summary, we provide the following contributions:

- We propose RealSWATT, the first software-based remote attestation framework for critical real-time devices that works on commodity off-the-shelf, low-cost embedded devices.
- We present the first attestation framework that exploits a separate processor core for attestation to ensure the correct scheduling and timing of real-time operations.
- We propose a new scheme called continuous attestation and a network architecture for the software-based attestation of embedded devices which allows us to tackle the strict timing constraints and hardware requirements of existing software-based attestation schemes [32, 241].
- Our framework allows to verify code- and data sections remotely to detect malware infection and malicious changes in configuration parameters.
- For our proof-of-concept implementation, we used one of the most popular IoT platforms, the ESP32 microcontroller, and conducted a detailed evaluation on a medical device, a syringe pump. We performed a full end-to-end example with an attack that compromises the syringe pump's configuration, which RealSWATT detects.
- We implemented RealSWATT into ESPEasy, a framework to use on real-world off-the-shelf IoT devices, and used it on different devices such as a smart plug, a smoke detector, and a smart light bulb.

3.1 Background: Software-Based Attestation

First, we briefly recapitulate on software-based remote attestation as a basis for RealSWATT. More on real-time embedded systems and remote attestation can be found in Chapter 2. In contrast to hardware-based and hybrid attestation approaches, software-based attestation schemes do not require special hardware features and can hence be used on commodity and legacy hardware, including IoT devices. The security of software-based attestation relies on the execution time of the software on the prover. Remote attestation is typically based on a challenge-response protocol. The verifier initiates the attestation by sending a request to the prover. This request typically includes a nonce to prevent replay attacks. The prover takes this nonce to perform a self-measurement, typically using a hashing function. Thereafter, the prover sends the result back to the verifier. Note that the verifier does communicate back to the prover. The prover is not aware of the result of the attestation.

The verifier measures the response time of the prover. If the response time is within a predefined margin, the measurement is assumed to be benign. A delayed response indicates that the prover has been compromised. Obviously, this requires precise measurement of the actual execution time and exact prediction of the expected execution [241]. Hence, this poses strict requirements for the implementation: the attestation process may not be able to be accelerated under any circumstance. Otherwise, the attacker can alter the result of the attestation process without violating timing constraints [57]. On the one hand, the attacker may not be able to alter the hardware, increase the processor speed, or extend the memory. On the other hand, the implementation attestation function must be optimal. Furthermore, it is important to cover all memory with the attestation, including unused memory: Any memory not covered by the attestation can be used by the attacker without detection. As the runtime of the attestation function is measured by the verifier, this measurement includes the time required for transmission of the attestation request and report besides the actual runtime of the attestation function. Therefore, it is necessary that also this transmission time can be precisely predicted to allow secure software-based attestation. In summary, there are significant challenges that pose strict requirements towards software-based attestation [32, 241]. Next, we describe those challenges in detail, followed by Section 3.4, where we explain how we address these in RealSWATT to enable secure software-based attestation of IoT devices.

3.2 Challenges

Detecting software attacks on devices in a connected system is a highly challenging task, in particular if the adversary has gained full control over a subset of devices. Even more so if the connected devices have high requirements with respect to their timing behavior due to the execution of real-time tasks.

Existing solutions to detect and report attacks in connected systems are either heuristic [94] or pose assumptions that are unrealistic for many critical real-time tasks [241]. Monitoring solutions at the network level are heuristic in nature and suffer from a high false-positive rate [297]. Hardware-assisted security mechanisms [159] rely on extensions and components, such as a TEE architecture or cryptographic co-processor (TPM), that are not available in the vast majority of deployed legacy embedded devices. Software-based attestation approaches target legacy devices that cannot provide a trust anchor; however, their approach to ensure the integrity of the measurement function inherently conflicts with the execution timing demands of real-time applications: Software-based attestation asserts the integrity of the measurement procedure by demanding all system resources to prevent the adversary from using free resources while precisely measuring its execution time. Besides the inherent conflicts when applied to real-time systems, software-based attestation is the only option (to enable heuristic detection of complete software compromise) for legacy systems. From the general challenges for secure remote attestation that we described in Chapter 2.5 we derive the requirements for a secure software-based attestation scheme for real-time embedded systems. Such a scheme poses a number of specific challenges to the underlying design and implementation:

Challenge 1: Real-Time Operation. A secure attestation scheme—without any trust anchor—running tasks with real-time execution requirements needs to overcome the inherent conflict between real-time execution guarantees and integrity guarantees for the measurement procedure to capture the device state of the prover.

Challenge 2: Parallel Tasks. Allowing the measurement procedure to respect the real-time demands of the systems' tasks could be easily misused by an adversary, for example, to restore a benign state while the measurement is performed [50]. Therefore, the measurement procedure must be able to capture the system state independent of the execution of real-time tasks.

Challenge 3: Roaming Adversary. Permitting the execution of potential malicious tasks in parallel to the measurement procedure provides the adversary with the option to dynamically adapt and move itself between memory areas, always restoring the currently measured section of memory.

Challenge 4: Network Transmission. Software-based attestation faces the challenge of accounting for the jitter in network transmission in remote scenarios, which prevents the verifier from precisely measuring the execution time. This leads to the following conflict. On the one hand, the verifier has to tolerate considerable time gaps to avoid false positives. On the other hand, an adversary might exploit these time gaps to manipulate the measurement and hide traces of an attack on the prover device.

Challenge 5: Time-of-Check/Time-of-Use. Remote attestation schemes, and in particular software-based attestation, face the time-of-check/time-of-use problem (TOCTOU) [298]. An attestation report only presents a snapshot of the prover's state. So the verifier learns no information whether the prover has been compromised (and restored) before the attestation or will be immediately after the attestation, thereby evading detection.

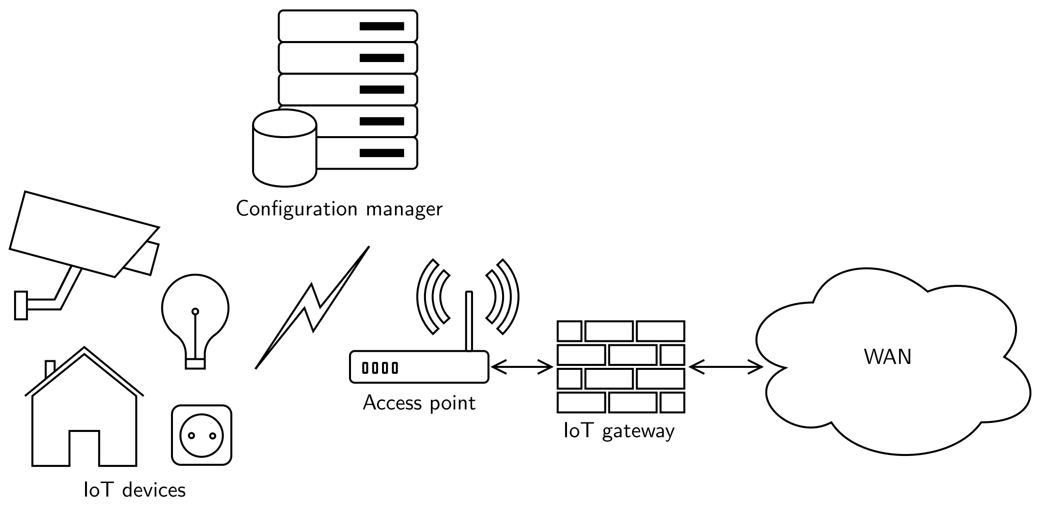


Figure 3.1: Network architecture of the real-time attestation approach.

RealSWATT overcomes these challenges by executing the measurement procedure on a dedicated CPU core while allowing the continuous execution of real-time tasks on another CPU core. Furthermore, our approach deploys novel techniques to tackle all challenges in order to design and implement a secure attestation solution.

3.3 Assumptions and Threat Model

RealSWATT attestation is designed to work with legacy IoT devices in real-world network environments. The network architecture is sketched in Figure 3.1. It targets scenarios with untrusted embedded systems running critical real-time tasks that should be attested by the remote verifier.

3.3.1 Assumptions

We consider a system of connected low-end embedded devices executing real-time tasks. We assume an untrusted system running tasks with real-time deadlines, the so-called prover, which is being attested by the remote verifier. Furthermore, we assume that the prover runs a real-time operating system (RTOS), ensuring correct scheduling and proper real-time operation. The RTOS is no mandatory requirement. However, it simplifies the integration of the RealSWATT framework into legacy devices. Without an operating system, the attestation logic has to be manually integrated, and it has to be ensured that the added methods do not influence the other tasks running on the device.

The system features a multi-core processor, of which one core is not utilized and is not required for correct real-time operation. The attested device is connected to a remote verifier via a wired or wireless network. There are no strict timing requirements towards the connection. We elaborate on bandwidth and timing requirements in Section 3.6.

All devices of the network are known to the verifier device. Therefore, they can communicate with each other and the verifier directly. However, all communication with external entities is routed through a gateway. We assume an IoT network structure consisting of multiple IoT devices that are being attested, a trusted verifier, and an IoT gateway for external communication, as sketched in Figure 3.1. The IoT gateway monitors external communication to detect abnormalities to prevent offloading of attestation tasks. Offloading remote attestation tasks to an external party requires frequent communication to a dedicated remote instance which clearly differs from the normal behavior of IoT devices. Monitoring traffic by its origin, goal, packet size, frequency, or content is a common method to secure internal or dedicated IoT networks. For example, the National Institute of Standards and Technology (NIST) suggests using such gateways to secure communication of embedded devices [44, 263]. In recommendation ITU-T Y.2060 [149], the International Telecommunication Union (ITU) also considers such gateways for IoT networks.

The configuration manager, as shown in Figure 3.1, is a common component in modern IoT architectures. It keeps track of the devices and their applied configuration. Configuration management software for IoT devices is commercially available [250]. Recent research also considers using such a configuration manager to set and organize security features on IoT devices on demand [69]. With the help of such a configuration manager, attestation can be enabled on a large scale without the need to set up attestation on each device individually. A further benefit is that the configuration can be easily provided to a verifier and included in the attestation reports.

3.3.2 Threat Model

Figure 3.2 shows the threat model. The adversary can compromise all embedded devices in the network via software attacks. The adversary knows the benign state and configuration of every device. It can observe all network communication. The adversary is able to modify program data as well as configuration data. Furthermore, the adversary can compromise an embedded device at each point in time as well as restore a device's benign state at any point in time.

The verifier and the gateway are assumed to be immune. Thus, the adversary cannot compromise them. All communication, except between devices and the verifier, is routed through the gateway. The adversary cannot introduce additional devices into the network. The hardware of the devices cannot be modified or manipulated

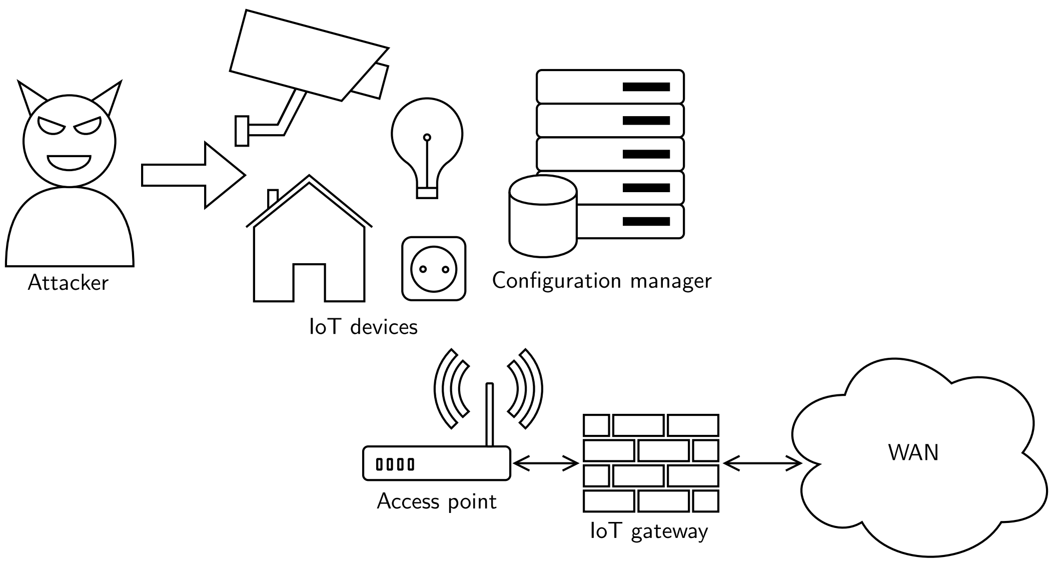


Figure 3.2: Attacker model of RealSWATT. The attacker can compromise all attested devices but can neither modify the configuration manager, which takes over the role of the verifier, nor the IoT gateway.

by the adversary, for example, by being enhanced by a more powerful device with more memory or a faster processor.

3.4 Concept of RealSWATT

Our design of RealSWATT introduces two new concepts:

1. Using a separate, idle processor core for attestation to separate the normal operation and the attestation tasks from each other, and
2. Continuous attestation, that is, attesting the system continuously during runtime.

Multi-core processors have many advantages in processing speed and energy consumption [118] and are becoming increasingly widespread [81], even if the development of real-time applications for multi-core processors is challenging [230, 286]. We leverage a multi-core processor design, which is nowadays commonly available on popular IoT platforms, but often only partially utilized.

We observed that in many IoT devices and especially real-time systems with a multi-core architecture, not all cores are fully utilized. Moreover, in some specific application areas, such as avionics systems, there are even regulations that limit the

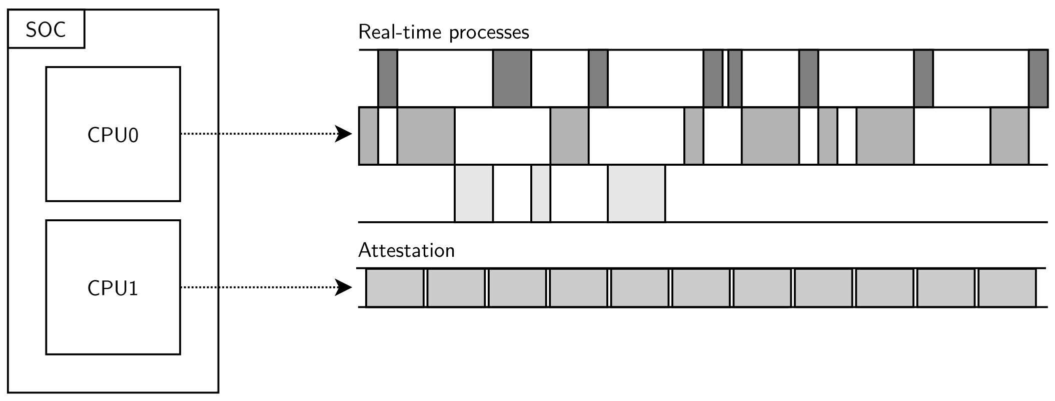


Figure 3.3: A system with multiple processor cores can use one core for real-time processes and one core to run the attestation in parallel.

use of additional cores for real-time operation [60]. In real-time computing, there exist several security frameworks that use a separate processor core of a multi-core system to implement new security features [245, 295]; for instance, Yoon et al. [295] utilizes it for intrusion detection and other follow-up works leverage it to cover memory usage [296] and analyze system call traces [294].

Software-based attestation (SWATT) relies on the precise prediction of the response times of the attested device. The verifier sends a challenge and measures the response time of the prover, which includes the execution time of the attestation and the transmission time. Hence, SWATT also relies on direct and undisturbed communication between the prover and the verifier. That is, if the response to an attestation request is delayed, the prover cannot distinguish between a false alarm caused by a transmission delay and an attack. For the latter, the delay is caused by the attacker covering their traces.

However, the assumption of undisturbed communication is unrealistic in practice. Nowadays, IoT devices communicate via wired or wireless networks that are shared with many other devices. As a result, these devices influence each other's transmissions, especially in wireless networks like Wi-Fi. Wireless networks inherently use a shared medium that is not only shared between the devices within the network but also with all other devices using the same frequency band. Hence, traditional software-based attestation [241] cannot be applied to these communication networks. We tackle this limitation by developing continuous attestation that eliminates the transmission time from the timing measurements of the attestation. We leverage this by continuously running the attestation such that the verifier can safely assume that, at all times, the prover is running an attestation. To do so, we use a dedicated processor core for the attestation.

Separate Processor Core for Attestation. Figure 3.3 shows the distribution of the real-time tasks and the attestation on different processor cores. The RealSWATT framework requires at least two processor cores but supports more cores without any changes. A single processor core is selected to execute the attestation. Both the attestation runs, and real-time jobs are time-critical. Missing timing deadlines for normal operation results in device malfunction as real-time properties are not met, and timing problems for the attestation make the verification fail, as the verifier assumes the device to be compromised as it does not respond in time.

Continuous Attestation. As a dedicated processor core can solely be used to perform the attestation, this allows the introduction of continuous attestation, where attestation constantly runs in the background. In traditional software-based attestation, (SWATT) [241], the proof of integrity of the prover code is based on the response between sending the attestation request and receiving the attestation report of the prover. Typically, the attestation request contains a nonce to ensure freshness and prevent replay attacks. Consequently, the transmission time between the verifier and the prover must be included in the timing assumptions, making SWATT impractical for all communications with varying transmission times, such as wireless networks or the Internet.

Our continuous attestation relaxes these timing assumptions as the attestation is constantly running so that even though the response is delayed, the verifier can safely assume that the prover has been running the attestation task. In RealSWATT, the verifier sends a new nonce while the prover is not yet finished calculating the attestation report for the verifier (Section 3.4.2). So, the prover can continue with the next attestation request directly after finishing the last one. We call this attestation method continuous attestation as it removes the gap between attestation runs. In contrast to the communication delay, the time required for the attestation can be determined precisely. In Section 3.6, we measure the runtime of attestation processes.

In the following, we describe the challenges that emerge when implementing a continuous attestation scheme.

3.4.1 Design Considerations

While using a separate processor core for remote attestation seems like a straightforward solution, it requires careful design decisions to ensure coverage of various security aspects. Software-based attestation has many strict requirements that have to be fulfilled to reliably verify the prover besides the accurate timing of the responses. Therefore, it is of utmost importance that an attacker cannot accelerate the attestation run itself. There are multiple ways how an attacker could speed up the attestation. Each of them has to be addressed accordingly.

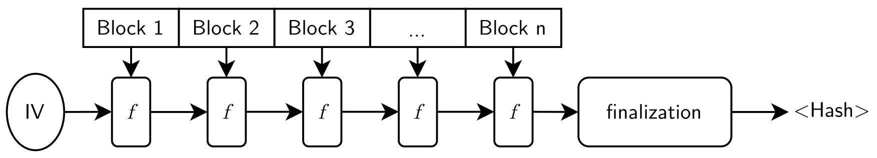


Figure 3.4: Merkle–Damgård construction of hash values. The process cannot be parallelized for speedup.

Parallelization. An attacker can potentially use all processor cores for attestation and ignore real-time-critical jobs, while genuine attestation can only use a single processor core. Suppose the attacker is able to speed up the execution of the attestation function. In that case, the attacker can circumvent the timing checks that are based on hardware limitations and potentially evade the remote attestation.

Therefore, the attestation scheme must be designed such that an attacker cannot benefit from multiple cores. Further, attestation relies on a hashing function. The hashing method must be designed to not be accelerated by parallelization, for instance, by using multiple processor cores in parallel. We tackle this challenge by using a Merkle–Damgård construction [182] as this popular hashing method fulfills this requirement. The functionality is shown in Figure 3.4: The process starts with an initialization vector. Then, the hash is calculated by adding block by block, where the next block is added in each step. In order to add the next block, the previous result is taken as input. This is a strictly sequential process. Hence, the process cannot benefit from parallelization or multiple processor cores [9]. Popular hashing methods using the Merkle–Damgård construction are, for instance, SHA-1 and SHA-2 [192].

Optimality of Hash Function Implementation. As the security of software-based attestation relies on the computational capability and timing threshold, that is, the execution speed of the attestation function, it must be ensured that the implementation of the attestation function is optimal and cannot be significantly accelerated. Otherwise, if the attacker is able to generate a valid hash, the saved time can be exploited for malicious activity. RealSWATT addresses this challenge by leveraging built-in hardware modules if available (as is the case for our target architectures) or well-studied hash algorithms. RealSWATT is not limited to a certain hash function: any secure hashing method that fulfills the Merkle-Damgård scheme is suitable for our attestation approach. For example, the popular SHA-2 hash function fulfills this requirement, which we also use in our implementation in Section 3.5 and case study in Section 3.6. For platforms without hardware support, we study their security regarding attacks against the hash function in detail in Section 3.7.3.

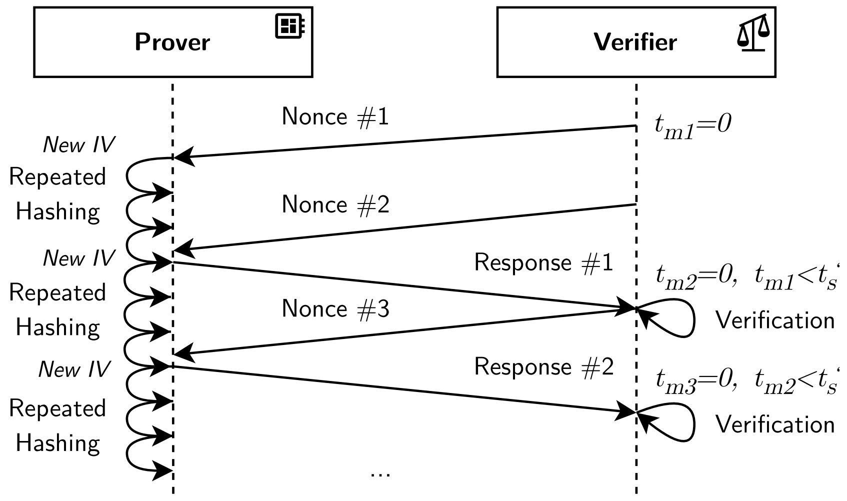


Figure 3.5: Protocol of the attestation process.

Empty Memory. As memory that is not covered by the attestation process could be used by an attacker, all executable memory has to be included in the attestation. Furthermore, an attacker could compress the data stored on the device to free up memory, which can then be used to store malicious code. REALSWATT prevents this as its continuous attestation constantly monitors all executable memory. As shown in Section 3.6, continuous attestation induces strict timing requirements. Deviations, for example, due to the need for decompression, make the attestation fail.

Offloading. An attacker could also offload attestation work to another device. In our attacker model in Section 3.3, we describe a remote attacker and exclude the scenario in which a local attacker is able to introduce more computing power into the attested device. However, the attacker could offload the attestation task to another powerful device, thereby breaking the attestation scheme. Due to the longer and varying transmission time, the verifier would not detect this. To tackle this issue, we introduce an IoT gateway that monitors all traffic from and to the network with the attested devices. Such security gateways are a common measure in commercial and industrial networks. But such filters can also be added to routers for small business and home networks. We elaborate on this network architecture in Section 3.4.3.

In the following, we will explain the attestation scheme and the network architecture in detail.

3.4.2 Attestation Scheme

As mentioned in Section 3.4.1, we needed to consider and evaluate several aspects of our design to create a practical software-based attestation approach for real-time embedded devices. Common and more advanced attestation methods like control-flow attestation [3] are not applicable as they either interfere with the runtime (instrumentation), which conflicts with real-time constraints or require additional hardware like TrustZone. Thus, with RealSWATT, we attest code and data regions of those legacy devices. A device can have multiple partitions containing the executable code and data, including device configuration. We hash those dedicated memory areas based on the protocol shown in Figure 3.5: The verifier sends a nonce to prevent replay attacks to the prover. The prover uses this nonce as an initialization vector for the hashing algorithm. Hence, the attacker cannot start the attestation before the nonce is known. Next, the prover calculates the hash of the memory region that has to be attested, for example, code or data sections. This concept is a common and reliable method for remote attestation [49, 91, 201, 241]. Finally, we read all data from the attested partitions and feed it either to the available hardware hashing module or into the optimized hashing algorithm. Usually, all code and data sections are combined and hashed. Thus one single hash value represents the code and data integrity of the device. It is also possible to limit the hashing only to certain memory sections. This option, however, should be used with care as it limits the appropriate state representation of the embedded device.

The hash is then repeatedly computed and returned to the verifier. The hash calculation is chained, and previous hash results are fed into the next repetition. Finally, the verifier measures the time t_m between sending the nonce and receiving the response. If the measured time is below the expected threshold time t_s, thus $t_m < t_s$, the device is assumed genuine. Otherwise, it has potentially been compromised. This process is continuously repeated to ensure that any compromise or malicious modification of the device is being detected. Therefore, the hashing function must have a predictable runtime. If the runtime varies, this allows the attacker to shift tasks to get computation time. This remains undetected, as the verifier has to assume the worst-case runtime. Therefore, the determination of an appropriate threshold is a key feature of software-based attestation. In Section 3.6, we measure the execution time for the attestation of a real-world device and describe how the threshold time t_s can be determined.

In a simple attestation protocol, there exists a gap between sending the resulting hash and receiving the next nonce, which consists of the network transmission time and the time of the verifier to send the next nonce after verification. In order to close this time gap between two successive attestation requests, in our solution, the verifier sends a second nonce while the prover is still processing the previous attestation request, typically by computing a hash. The second nonce is received and temporarily stored in a queue. This allows the prover to continuously process attestation requests and removes the impact of network delays.

To do so, the verifier has to send the nonces such that even under worst-case network latency, the next attestation request arrives before the previous attestation run is finished. Sending two nonces without delay allows an adversary to compute the hash on other cores simultaneously. Therefore, it is important to send the hash just in time. Given a time t_{att} to complete the attestation and a network latency t_{rtt}, the verifier has to send the next nonce $t_{att} - \max(t_{rtt})$ to ensure that the next nonce arrives on time. Note that it is required $t_{att} \ggg t_{rtt}$ to guarantee correct attestation.

The nonce sent by the verifier serves both as a new initialization vector and as a synchronization point. In a scenario with a long-running attestation task with a one-time initialization only, side effects like clock skew between devices would come into play. We reliably synchronize the verifier and the prover by continuously sending nonces as new initialization vectors.

The verifier checks the interval in which the results are returned from the prover. If results are delayed or missing, a compromise can be assumed. The verifier can react accordingly, for example, by raising an alarm or rebooting the prover to return to a trustworthy state.

This process, where the prover saves the next nonce in advance, makes our attestation scheme independent from the transmission time between the prover and the verifier. Even variances in the transmission time do not pose any problems as long as the transmission time is significantly shorter than the time required for the attestation $t_{att} \ggg t_{rtt}$. It is possible to configure the runtime of the attestation by repeatedly executing the hashing function: the result of the hashing is used as the initialization for the next hashing. So, a long non-parallelizable row of executions is generated. This makes it possible to adapt the attestation's duration to the actual transmission time requirements and consider the device's processing speed. Because a potential attacker cannot offload the computation to an external device, and we carefully choose the time intervals for attestation requests (sending the nonces), intercepting the next nonce does not provide any benefit to an attacker. In Section 3.6, we elaborate on how the attestation time can be configured using a real-world example. Suppose the worst-case transmission time is significantly shorter than the execution time of the attestation. In that case, the next nonce can safely arrive at the prover before the previous attestation process finishes. Thus, there are no time gaps between successive attestation requests. We call this approach continuous attestation, which is also a key aspect in enabling practical software-based attestation.

Furthermore, our attestation method has several benefits with regard to existing legacy embedded devices. Our attestation protocol is lightweight with a nonce of 4 byte and an attestation report of 32 byte. Thus, it causes only a slightly increased network load and is suitable for low-speed IoT networks, as discussed in Section 3.6.3.

Another aspect of RealSWATT is its real-time capability and ease of integration, which we evaluated on real-world devices in Section 3.6.5. As already shown in Figure 3.3, we exploit the availability of a second core to handle the attestation process in parallel to real-time operation. While the use of a second core allows maintaining real-time capability, it comes with its own set of challenges, such as the parallelization of the hash computation by two cores, which we discussed in Section 3.4.1 and Section 3.7.

3.4.3 IoT Network Architecture

The usage of a dedicated network architecture allows reliable and secure software-based attestation with varying transmission times as described in Section 3.3. It consists of several parts: the attested IoT devices, the central configuration and attestation server, and the IoT gateway. In addition, this common network architecture allows to prevent offloading attacks, thus moving the attestation task to external devices.

Like many IoT devices nowadays, the attested devices communicate over wireless communication (Wi-Fi, IEEE 802.15.4/ZigBee[1], Z-Wave[2]) with the configuration and device management server. Therefore, there are no strict requirements towards connection speed, transmission times, or jitter. In the RealSWATT attestation scheme, the verifier is implemented in a central device and configuration management server. This configuration and device management server keeps the current configuration of all IoT devices and also performs the verification of these IoT devices. Hence, it is possible to include individual configurations for each device and also detect modifications in these configurations. The IoT gateway monitors external communications and prevents a corrupted device from communicating with external entities to offload the attestation routine and hence break the attestation. If the attestation request can be sent to an external instance, this so-called offloading attack might invalidate the result. This can be achieved, for instance, by monitoring messages and response times. As attestation requests need timely responses, this leads to suspicious transmission patterns. Both the central device and configuration management as well as IoT gateways are commonly deployed in real-world networks as previously discussed in Section 3.3.1.

3.5 Implementation

We implemented RealSWATT on commercial off-the-shelf hardware to show its general applicability. The prover was integrated into FreeRTOS, which is a popular

[1]`https://zigbeealliance.org/`
[2]`https://z-wavealliance.org/`

real-time operating system [116]. The verifier was implemented on a Raspberry Pi running Linux. The verifier can also be implemented on other devices, such as commodity X86 computer systems. The only requirement is a connection to the IoT network and enough computing resources to handle and verify the attestation requests. We are using raw UDP packets for communication in order to reduce the side effects of network transmission and minimize communication overhead.

We integrated the prover for RealSWATT into different IoT devices to show its broad applicability: a syringe pump, a smart plug, a smoke detector, and a smart light bulb. For the smart plug, smoke detector, and smart light bulb, we have used a framework called ESPeasy[3]. It allows the generation of alternative firmware images for a wide range of off-the-shelf IoT devices powered by the ESP32 and includes code for peripherals such as the smoke sensor. However, even without this framework, integration into existing off-the-shelf devices is generic and straightforward as described in Section 3.5.4.

To evaluate the functionality, we integrated all components into a real-world test bed consisting of the typical components of an IoT network. We tested the RealSWATT attestation using a complete end-to-end example consisting of a device being monitored by the verifier. The attested device is then compromised, which the verifier instantaneously detects.

In the following, we describe the main components of the implementation. First, we explain how we implemented the prover and the verifier. Then we present the test bed we developed to test the real-world applications. Finally, we show the necessary steps to integrate RealSWATT into new applications.

Please note that this section only gives a general overview of the implementation. Implementation details like timing thresholds need to be fine-tuned for typical embedded devices and their networks. We evaluate and provide these details for our test bed in Section 3.6.

3.5.1 Prover

We use the Espressif ESP32 system-on-chip (SoC), which is a popular component of typical IoT devices, for example, smart light bulbs and power plugs [96] as it also integrates Bluetooth and Wi-Fi modules.

We have implemented the RealSWATT remote attestation method using the popular FreeRTOS real-time operating system. FreeRTOS can manage multiple processor cores and allows the attachment of processes to a dedicated core. The scheduler then does not move the attached processes across cores. Instead, we dedicate one core to the attestation process. We have implemented the RealSWATT remote attestation method using the popular FreeRTOS real-time operating system on

[3]`https://espeasy.readthedocs.io/en/latest/`

the ESP32. The ESP32 has two Tensilica Xtensa processor cores [97]. Since the attestation is scheduled on a dedicated core, the attestation does not interfere with the real-time operation. Real-time operation, as well as attestation, are handled by different cores and in parallel.

RealSWATT performs static attestation covering code and data memory of the prover. This is achieved by including the program and configuration data partition in the attestation requests. The hashing is performed using the mbedtls[4] library, which also supports hardware-supported hashing on the ESP32. In order to prevent replay attacks or the usage of pre-computed results, each attestation run is initialized using the nonce provided by the verifier. Continuous attestation is realized using a queue. When the prover receives a UDP packet containing a nonce from the verifier, the nonce is written to a queue of limited size until the prover handles it. This way, attestation runs are executed seamlessly after each other. The attestation result is returned as a UDP packet to the verifier.

The attestation is implemented as two separate tasks, one receiving the attestation requests from the verifier and one task performing the actual attestation. Both tasks are pinned to the dedicated and previously unused processor core to ensure no side effects between the attestation and the real-time operation of the attested device. We verify that the attestation runs continuously without distractions from the real-time operation. RealSWATT attestation has the possibility to configure the runtime of the attestation. The runtime needs to balance between delay, that is, the time until a compromise of the attested device is detected, and communication overhead. This balancing is achieved by configuring the number of repetitions of the hashing function. More repetitions invoke a longer runtime. At the end of each attestation run, the result is sent to the verifier, which then checks its validity. When integrating RealSWATT attestation into a new device, the runtime of the attestation function has to be determined, and the number of repetitions has to be configured. In Section 3.6, we perform a detailed measurement of runtime and communication overhead on the implementation of RealSWATT.

3.5.2 Verifier

The verifier implements the RealSWATT attestation protocol described in Section 3.4.2, sending nonces to the prover and handling incoming attestation reports, such as checking the timing threshold and verifying the integrity of the measurements.

There are two available implementations for the verifier. First, we used Python, and later we opted for C++. Since Python is an interpreter-based programming language, the Python implementation of the verifier can be used without adjustment across a wide range of devices. The only requirement is that a Python interpreter

[4]`https://tls.mbed.org/`

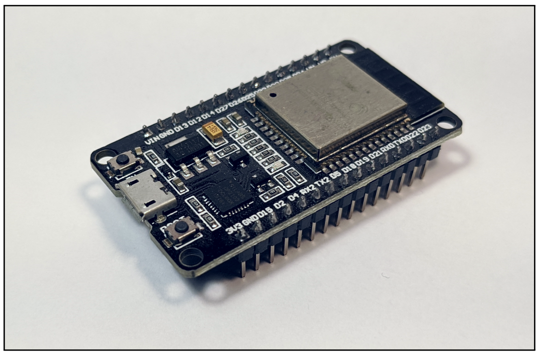


Figure 3.6: Photo of the NodeMCU ESP32 developer board.

for the device is available. However, we assumed a worse network response time than with a native C or C++ implementation. In order to check the influence of the programming language, we also implemented the verifier as a native C++ application. Contrary to our expectations, the programming language had little to no impact on the measured network response times.

As mentioned, the verifier receives the attestation reports and checks their validity. According to the RealSWATT attestation protocol, the verifier sends two nonces to the prover. Each nonce triggers an attestation run. It is the verifier's responsibility to time the transmission of these and all following nonces. The design of the protocol is explained in Section 3.4.2. In Section 3.6, we elaborate on how to correctly time these message intervals and determine the thresholds for the attestation.

3.5.3 Test Bed

To evaluate RealSWATT, we built a test bed of an IoT network as sketched in Figure 3.1 consisting of IoT devices, a Wi-Fi access point, and a verifier. The IoT devices were implemented on NodeMCU ESP32 developer boards[5]. A TP-Link TL-WDR4300 Wi-Fi router[6] running OpenWRT 19.07.7 was used as IoT gateway and a Raspberry Pi 3+[7] with Linux was running the C++ implementation of the verifier. Figure 3.6 shows a NodeMCU ESP32 developer board. The setup was located in an office environment during workdays with frequent Wi-Fi usage. The Wi-Fi access point provided a separate IoT network in a 2.4 GHz range, as the ESP32 is only able to work within this frequency range. The test bed reflects typical usage in practice, for example, in hospitals or factories with different interfering Wi-Fi traffic and other wireless devices that influence the communication between the

[5]`https://joy-it.net/en/products/SBC-NodeMCU-ESP32`
[6]`https://www.tp-link.com/ch/home-networking/wifi-router/tl-wdr4300/`
[7]`https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/`

prover and the verifier. The influence of other wireless devices and Wi-Fi traffic and its implications for the attestation is further analyzed in Section 3.6.

3.5.4 Integration Guidelines

The integration of RealSWATT is straightforward. IoT devices often rely on a real-time operating system (RTOS) [34], which allows to manage and appropriately schedule multiple concurrent tasks on multiple cores. The usage of an RTOS gives standard interfaces and methods to add the attestation service. In order to integrate RealSWATT, we have created additional tasks. We added a UDP service for communication and an attestation task on the dedicated core. The usage of an RTOS is no requirement for RealSWATT. Integrating RealSWATT into devices with no operating system, so-called bare metal systems, is also possible. However, the integration will need to be performed much more carefully as one cannot rely on the abstraction and features provided by a real-time operating system.

In the next section, we evaluate RealSWATT and show its general applicability. To do so, we perform a case study and integrate RealSWATT into a medical device and an IoT framework.

3.6 Evaluation

In this section, we show that RealSWATT attestation is well-suited for real-world IoT setups and can be applied in practice. As described in Section 3.5, the RealSWATT attestation was deployed on different embedded devices. To show the general applicability of the RealSWATT attestation concept, we investigate its runtime and timing constraints. As elaborated in Section 3.4, timing is a crucial security factor in software-based attestation. We measure the response times in our exemplary syringe pump example and explain how timing thresholds for the verification of the attestation can be determined. We further analyze the overhead induced by the attestation. This covers both additional power consumption caused by the usage of the second processor core as well as the communication overhead for attestation requests and responses.

In a full end-to-end example, we show the functionality of the RealSWATT attestation by performing an attack on a vulnerable device that is detected by the attestation.

3.6.1 Timing Behavior of the Attestation Function

Timing is a crucial component for the security of software-based attestation. While in traditional software-based attestation, the response time to the verifier is the

Table 3.1: Measurement of the runtime of the attestation with and without the delay due to the Wi-Fi communication. All measurements are taken in ms.

Rounds	Type	Min/Max	Mean	Std. Dev.
0	Direct	7.979/8.14	7.988	0.032
	Network	36.066/226.36	72.207	11.43
1	Direct	9.752/10.265	10.004	0.038
	Network	49.502/279.684	225.76	20.84
2	Direct	19.748/20.276	20.006	0.057
	Network	69.367/274.108	225.96	18.81
5	Direct	50.005/50.005	50.005	0.0
	Network	59.662/276.617	224.61	23.26
10	Direct	100.004/100.004	100.004	0.0
	Network	125.149/329.905	228.32	9.952
20	Direct	200.004/200.004	200.004	0.0
	Network	217.307/439.912	241.46	46.96
30	Direct	300.004/300.004	300.004	0.0
	Network	320.665/469.364	431.11	16.91
50	Direct	500.004/500.004	500.004	0.0
	Network	518.593/726.847	637.27	13.01
100	Direct	1000.004/1000.004	1000.004	0.0
	Network	1019.64/1218.943	1048.91	13.15
150	Direct	1500.005/1500.005	1500.005	0.0
	Network	1517.224/1679.623	1661.35	11.12
200	Direct	2000.004/2000.004	2000.004	0.0
	Network	2009.749/2179.673	2072.29	9.24
300	Direct	3000.004/3000.004	3000.004	0.0
	Network	3009.71/3190.993	3095.92	8.67

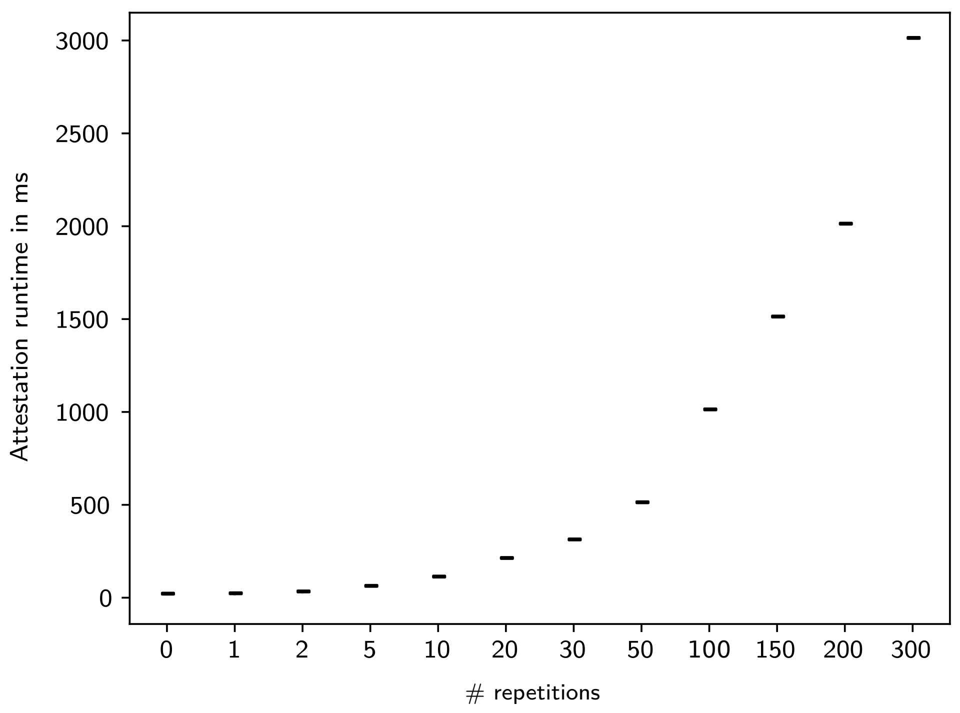


Figure 3.7: Runtime of the attestation process on the prover with a different number of repetitions.

relevant part of the security, in continuous attestation, the runtime of the attestation itself is important, while the transmission time can be neglected.

We performed a measurement study in our IoT test bed to determine the response times of the prover to the challenge depending on the number of repetitions of the hashing function and the variance of the transmission in the Wi-Fi network. In order to perform a reliable attestation, the number of repetitions of the hashing function has to be chosen such that its runtime dominates the variance of the transmission. The continuous attestation only works if the verifier receives the response of the prover before the prover finishes the next attestation request. A too-long attestation run increases the time span between two attestation reports of the prover. Hence, the delay between compromise and its detection becomes larger.

Table 3.1 shows the measured runtime of the attestation function, including and without the overhead due to the wireless network using different numbers of repetitions. In Figure 3.7, the runtime of the attestation process with respect to the

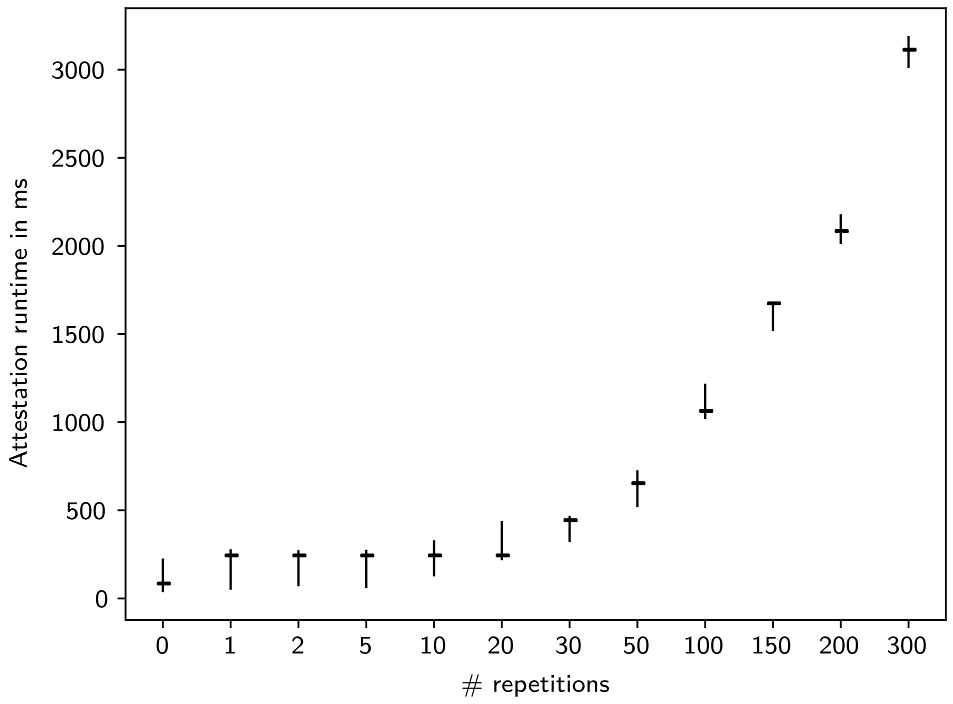


Figure 3.8: Response time of the prover with a different number of repetitions, this includes both the attestation runtime and the overhead due to the Wi-Fi communication.

number of hashing repetitions is plotted. In addition, since the hashing needs to dominate the variance of the transmission, we also plotted the response time with their respective variances in Figure 3.8 for direct comparison.

We conducted all measurements on the described test bed using the syringe pump implementation and repeated them to cover for any variations. We repeated the Wi-Fi measurements 600 times. The measurements without Wi-Fi were repeated 100 times due to their lower variability. The time including the Wi-Fi transmission, was measured on the Raspberry Pi. The runtime without the Wi-Fi overhead was measured on the ESP32 with its internal clock.

As expected, the variance of the runtime of the attestation without any communication is minimal. The highest standard deviation in the experiments was 56.83 µs in the case of only two repetitions. We measured no deviation in all cases with more repetitions, that is, more than five. This makes the implementation well-suited for software-based attestation, as strict timing limits can be selected. In comparison, the measurements which include transmission via Wi-Fi have much larger deviations, as Figure 3.8 shows. For example, in the case of ten repetitions, the time until the verifier gets a response from the prover varies between 110 ms and 303 ms. These results clearly show that such a Wi-Fi setup is inadequate to be directly used for traditional software-based attestation.

The values in Table 3.1 can be used to find optimal parameters for the attestation. To select suitable parameters, the minimum and maximum values of the attestation can be compared to find the optimal compromise between the runtime and the delay until a compromise is detected. These parameters directly influence the required number of communication events for the attestation. These parameters can also be used to configure the timeout thresholds for the verifier to detect delays in the attestation. All attestation runs have about the same execution time, and the variance between the executions is negligible. In contrast, the time until the verifier actually receives the attestation response varies widely. A software-based attestation without the RealSWATT continuous attestation approach is not feasible under these circumstances. Based on these results, it is possible to determine the adequate number of repetitions for the given use case. In the case of the syringe pump, we opted for 100 repetitions, yielding attestation reports in about 1 s intervals. These measurements are also required to configure the verifier to detect malicious behavior resulting in timeouts as the response time varies. The measurements show that RealSWATT is capable of working on IoT devices with wireless communication in practice.

3.6.2 Power Consumption

The second important aspect for real-world deployment of RealSWATT is power consumption. Continuous attestation causes constant additional computational tasks for the attestation core, which results in increased power consumption. Often, IoT devices are battery-powered and are expected to have long battery lifetimes. For instance, a smoke detection sensor in the corner of the ceiling runs years without battery replacement. Therefore, power consumption is also a concern of such IoT devices.

Consequently, we conducted a case study and measured the power consumption of the syringe pump with and without the attestation running. Without attestation, we have measured an average consumption of 46.2 mA. With attestation, we measure a slightly increased power consumption of 46.8 mA. So, attestation accounts for an increase in power consumption of 0.6 mA in this case, about 1.3%.

In order to evaluate the measured power consumption, we have checked the corresponding data sheet [103] of our ESP32 evaluation board. The data sheet provides the expected power consumption with respect to the operating mode of the chip for both the processor and wireless module. The wireless communication module dominates the power consumption of the ESP32. Its power consumption varies depending on the transmission mode. It ranges from 95 mA and 130 mA to receive and transmit via Bluetooth; up to 100 mA to receive and 240 mA to transmit via Wi-Fi IEEE 802.11b/g/n. However, this power consumption is only present during the regular send and receive intervals and thus needs to be treated as a peak power consumption.

The power consumption of the CPU depends on its operating frequency and overall usage of the CPU. We have configured the syringe pump to operate at the full CPU clock of 240 MHz for test purposes. This CPU frequency has the highest difference between no CPU usage (30 mA) and full CPU usage (68 mA). The naive assumption would be at least a 50% usage of the CPU (full utilization of the second core) for attestation, resulting in significantly higher power consumption. However, our implementation used the integrated hardware hashing unit, which is more power-efficient compared to a software-based calculation of the hashes.

We furthermore observed that the additional network traffic for the attestation is negligible for devices that already communicate on a regular basis. The syringe pump in our case study provides a remote command interface. Thus, the wireless communication module of this device is already in use. So sending and receiving attestation messages only slightly increases the overall power consumption.

3.6.3 Communication Overhead

Another important aspect is the amount of communication required for the attestation. First, IoT devices often use wireless communication, which is a shared resource with a limited frequency spectrum. With multiple devices communicating via the same channel, the network latency, as well as package drops, increase. Wireless communication is similar to traditional bus communication in that regard. Packets sent simultaneously collide and need to be retransmitted. Each wireless transmission takes a portion of the available bandwidth. For the RealSWATT remote attestation protocol, we only require the transmission of the hash value with 32 byte for the attestation reports and 4 byte nonce as attestation request. The transmission frequency can be configured, as discussed above, between several attestation requests per second to one every few seconds. These low demands make RealSWATT attestation suitable for working with low-bandwidth transmission protocols. For example, the popular IoT wireless protocols Zigbee and Z-Wave have transfer speeds ranging from 20 kbit/s up to 250 kbit/s [251]. Even the lowest transfer speed is sufficient to successfully run RealSWATT attestation with reasonable attestation frequencies.

3.6.4 Race Conditions

Attestation and real-time operation run in parallel on two dedicated cores. However, the attestation process requires access to the application memory to check for malicious activity. Thus, even when both operations are executed on a separate core, resources still need to be shared, which could lead to a potential race condition. In practice, race conditions between real-time operation and attestation rarely occur. This is because most embedded applications focus on GPIO (General-Purpose Input/Output) and thus have little memory interaction. Memory access can be prioritized depending on the attestation goals and type of real-time application. Most embedded devices in the domain of soft real-time systems will tolerate infrequent deadline misses. Thus, the attestation can potentially be prioritized in these cases. Hard real-time systems are usually strongly tight to the outside world and very GPIO intensive. As a result, memory accesses on hard real-time systems are very short. The acceptable delay for the attestation can be set to a value that will detect malicious activity but allow short delays caused by the real-time application. This value $t_{rtdelay}$ is distinct for each embedded device and application context and should satisfy: $t_{attack} \geq t_{hash} + t_{rtdelay}$. The delay $t_{rtdelay}$ is determined by the longest operation the real-time application would perform on the flash memory: These are often quick reads of configuration data (for example, the amount of medicine for injection on the syringe pump).

We have implemented RealSWATT into systems with hard real-time requirements like medical devices (syringe pump) and common IoT devices like smart plugs, see Section 3.6.5. We evaluated the impact of race conditions in these settings and found that they are highly unlikely and do neither influence the real-time requirements nor the attestation.

3.6.5 Implementation on Real-World Devices

To show the applicability of our approach to real-world applications and deployments, we applied RealSWATT to a medical device with strict real-time requirements and integrated RealSWATT attestation into an open-source firmware for IoT devices.

The first use-case is a syringe pump [285], a medical device that injects medication into a patient at a defined time interval. Hence, a syringe pump provides critical functionality and has strict real-time requirements. This open-source implementation of a syringe pump has already been used in previous works (see C-FLAT [3] and HERA in Chapter 6) to show the feasibility of control flow attestation and hotpatching of real-time devices.

In addition, we implemented RealSWATT on top of ESPEasy. ESPEasy is an alternative popular open-source firmware that allows replacing the firmware of existing IoT devices like smart plugs or temperature sensors. ESPEasy supports a

Listing 3.1: Default ESP32 partition layout without over-the-air updates [102]

```
# ESP-IDF Partition Table
# Name,    Type, SubType, Offset,  Size,Flags
nvs,       data, nvs,     0x9000,  0x6000,
phy_init,  data, phy,     0xf000,  0x1000,
factory,   app,  factory, 0x10000, 1M,
```

Listing 3.2: Internal configuration of the syringe pump

```
typedef struct {
    uint16_t syringe_volume_ml;
    uint16_t syringe_barrel_length_mm;
    float threaded_rod_pitch;
    ....
} internal_settings;
```

wide range of different devices and even extends their functionality. By implementing RealSWATT on ESPEasy, we proved that a wide range of legacy IoT devices can be easily secured through our attestation method.

In the following, we explain real-world details of the attestation based on the syringe pump use case implemented on the ESP32. The ESP32 allows for the custom creation of partitions. A developer can define memory sections on the chip in a data structure called partition table [102]. The partition structure depends on the implemented application and the required functionalities. For example, additional partitions are needed if an update mechanism such as over-the-air (OTA) update [100] is used. A simple application with no OTA update functionality consists of the three partitions as listed in Listing 3.1.

By default, the ESP32 uses three partitions, which contain the program and configuration data. In particular, the partitions are used for the following tasks.

`factory.` This partition contains the application code, that is, the executable.

`phy_init.` The second partition contains data required for the physical initialization process of the device.

`nvs.` The third partition stores the actual configuration of the application.

Listing 3.3: Medical configuration of the syringe pump

```
typedef struct {
    uint32_t injections_ms;
    uint16_t dosage_ml;
    uint8_t bolus_step_index;
} medical_settings;
```

The syringe pump is implemented with this default partition mapping. The code is saved on the factory-partition, and the configuration data is included in the nvs-partition. The syringe pump comes with multiple internal and external configuration options. Listing 3.2 shows the internal configurations that cover the physical characteristics of the syringe pump, such as the length of the syringe barrel. The internal configuration is required to transpose configured information, such as the amount of medicine, into the precise amount of rotation steps of the stepper motor driving the threaded rod of the pump. The syringe pump also has its usual medical settings available to the medical personnel, such as injection intervals and amount of medicine, as shown in Listing 3.3.

In order to attest the syringe pump, the data from all three partitions factory, phy_init, and nvs is read and concatenated. Then, we append the nonce and feed this data into the hardware hashing module of the ESP32. The resulting hash value is then repeatedly rehashed and sent to the verifier. Since the verifier knows the original syringe pump code as well as the physical initialization parameters and the configured options, it can verify the correct state of the syringe pump. The verifier can either integrate a device configuration manager or be notified by an external one about legitimate configuration changes. In our use case, we have integrated this functionality into the verifier.

The verifier sends the next nonce so that it arrives at the prover just before the end of the expected attestation time, even with worst-case network latency. As explained in Section 3.4.2, we have chosen 100 repetitions, such that $t_{att} = 1000ms \ggg t_{rtt}$ as Table 3.1 shows. The verifier is configured to send the second nonce 750 ms after the previous nonce.

3.6.6 End-to-End Case Study

To show the full capabilities of RealSWATT, we developed a full end-to-end example: a vulnerable real-time device that is being monitored. The vulnerability is used to compromise the device. This is then detected by the verifier. To do so, we have integrated a common vulnerability into the syringe pump: an insecure configuration interface, as the most common attack vectors of IoT devices are weak, guessable, or hard-coded passwords and insecure network interfaces and services [208]. In the case of ESPEasy, the web interface is only reachable via plain HTTP. Hence a passive man-in-the-middle (MitM) attack can easily be used to obtain passwords or authentication tokens [72]. Especially the usage of wireless interfaces further eases MitM attacks. Furthermore, per default, the login process of the web interface does not have any rate limiting, allowing efficient brute-force or dictionary attacks, for example, using hydra[8].

For our proof-of-concept (PoC), we hijack the command interface of our medical device and send a malicious configuration to the unit. This could trigger a buffer overflow and launch a more sophisticated attack or just manipulate the configuration. In the case of the syringe pump, even a configuration change could lead to lethal consequences for the patient: With our attack, it is possible to arbitrarily modify the amount of injected medicine. As soon as these changes are applied, the configuration on the nvs-partition is updated. At the latest, in the next attestation run, the hash value of the nvs-partition changes, which is sent in the attestation report to the verifier. The verifier determines that the configuration differs from the intended configuration and raises an alarm. For more details on the partitions and the implementation of the attestation, see Section 3.6.5.

3.6.7 Summary

In our practical evaluation of RealSWATT, we have shown that the attestation runtime can be adjusted by hash repetitions to dominate the variances in network response times of heavily used wireless networks. Consequently, our proposed attestation method for legacy devices is feasible for wireless IoT networks. In addition, we have measured only a slightly increased power consumption from 46.2 mA to 46.8 mA per hour for attestation. The increase is so small due to the commonly available hardware hashing unit's use, which reduces the second core's workload. Furthermore, as most IoT devices already use the wireless communication module on a regular basis, the additional power consumption for wireless communication is also minimal. Concluding, RealSWATT is suitable for application in real-world IoT devices.

3.7 Security Discussion

The RealSWATT attestation framework uses several new techniques to perform reliable software-based attestation of critical real-time embedded devices. The security of software-based attestation is based on multiple premises, which all have to be fulfilled in order to guarantee the integrity of the attested device. In the following,

[8]`https://github.com/vanhauser-thc/thc-hydra`

Variable	Definition
t_s	Threshold time
t_m	Message response time
t_{rtt}	Round trip time
t_{att}	Attestation runtime (all runs)
t_{hash}	Hashing runtime (single run)
t_{attack}	Minimal runtime for attack
t_{write}	Time of the flash to write to the sector

Table 3.2: Definition of times for security analysis.

we discuss the formal criteria for the attacker to stay undetected as well as possible attack scenarios, including mitigations.

Attack Model. To prevail against malicious activity, attackers must perform their attack and hide all traces before the attestation can detect those changes by means of hashing the memory. We define the different time variables to analyze diverse attack scenarios in Table 3.2.

A successful attack must satisfy $t_{attack} < t_{hash}$, which translates to a scenario where an attacker completes the attack before a check can be performed (single hash iteration with time t_{hash}). Further, an attack is only successful if the attacker is able to either manipulate the program code or configuration data stored in the flash memory (given the threat model defined in Section 3.3.2). As a result, the attacker needs to rewrite at least one flash page, which takes t_{write} time.

The attack scenario is depicted in Figure 3.9. t_{hash} denotes the runtime of the hashing of the complete memory. Every n repetitions, the result is sent to the verifier in the attestation report. One hash iteration covers all n flash memory blocks b, that is, $b_0 - b_n$, starting at a random location r. To prevent replay attacks, we integrate a nonce *nonce* provided by the verifier: $hash = \{nonce, b_r...b_n, b_0...b_{r-1}\}$. Multiple hash iterations are concatenated so that instead of the nonce, the hash result of the previous iteration is used as the starting point: $hash_i = \{hash_{i-1}...\}$.

Attack Requirement. Within the Attack Model (see Section 3.2), we have recapitulated the attestation procedure from the security viewpoint and introduced variables to describe the attacker's success conditions formally. The attacker's goal is to stay undetected. Obviously, the only feasible strategy for the attacker is to perform the attack before the attestation can detect it. As described above, this requires $t_{attack} < t_{hash}$. The time of an attack (t_{attack}) depends on the concrete attack scenario and cannot be exactly determined. However, a lower bound of t_{attack}

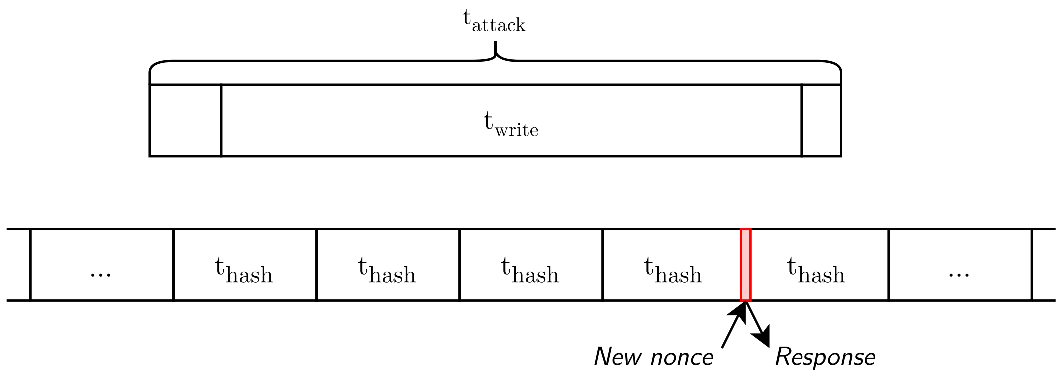


Figure 3.9: Attack scenario of RealSWATT. The device is attested in the background. The minimum attack duration is longer than the attestation runtime.

can be given due to hardware limitations such as the flash write time (t_{write}). We will elaborate on this in the following.

3.7.1 Hardware Restrictions

Embedded devices have hardware restrictions such as fixed times to write and read memory that the adversary cannot change (see Section 3.3.2). Since the attacker is not able to replace physical hardware, the attacker is bound by the hardware; especially, slow write operations to flash memory. This allows us to derive a lower bound for attacks (t_{attack}).

In case the device features a hardware hashing module, the attacker has to deal with fast hashing, thus short intervals in which the memory is attested. In addition, embedded devices provide flash memory as their main memory, and embedded devices such as the ESP32 typically rely on the external flash. Flash memory has to be written sector-wise, and flash typically uses a sector size of 4 kbyte. To erase one sector of a commonly used and quick flash chip takes about 50 ms. A similar fast flash chip is also deployed on our ESP32 boards [93]. Other flash chips require even 100 ms to erase a sector [92]. Since a successful attack requires at least one write, the minimal duration of any attack is t_{write}, so $t_{attack} \geq t_{write}$. The availability of hashing modules and flash speeds differ. Hence, hardware restrictions, in particular t_{write}, need to be considered for every device. In the specific case of our ESP32 board a sector write to the flash requires 50 ms [93] but a single hash iteration only takes 10 ms (see Table 3.1). As such, memory manipulation attacks are detected by RealSWATT for all modern embedded devices shipped with a hashing module.

3.7.2 Common Attack Scenarios

Remote attestation schemes and, in particular, software-based attestation schemes are vulnerable to several attack techniques.

Compressing Data on the Device. A compression attack is a typical technique to undermine software-based attestation [241]. The idea is to use compression mechanisms on the attested device to free up memory that is not covered by the attestation so that it can be used to store software that remains undetected from the attestation. In the worst case, the attacker decompresses malicious data on-the-fly at attestation time. Since RealSWATT targets multi-core processor architectures, the attacker has the capability to perform the attestation and decompression in parallel. However, a compression attack requires rewriting parts of the memory. Moreover, these writes are relatively slow. As these write accesses take more time than the interval t_{hash}, $t_{write} > t_{hash}$ still holds. Hence, we conclude that it is not possible to rewrite parts of the memory by means of a compressed version without detection.

In addition, RealSWATT deploys further techniques to prevent compression attacks. Since it is challenging to compress random data [241, 291], we fill free memory with random data provided and controlled by the verifier. The attestation reports include the random data, thereby allowing the verifier to detect modifications easily. Furthermore, due to continuous attestation, the device is attested frequently (every 10 ms as shown in Table 3.1). Thus, the attacker can only guess whether a data segment is currently being attested, reducing chances of success drastically.

Time-of-Check/Time-of-Use. It is a well-known problem that existing attestation schemes are susceptible to time-of-check/time-of-use (TOCTOU) attacks [254]. An adversary capable of restoring the memory in a given time frame before the next attestation will remain undetected. In contrast to known attestation schemes RealSWATT significantly reduces the adversary's success by leveraging the following:

1. The attestation continuously runs in the background, checking the complete memory regularly while the device operates normally.
2. The attestation starts from an arbitrary memory position (derived from the nonce). The attacker can neither predict nor influence this starting point. Halting the attestation core or process will be detected by timing thresholds.

Existing software-based attestation techniques like SWATT [241] interrupt normal operation for attestation. This is not an option for real-time systems, as these interruptions conflict with real-time requirements. As RealSWATT attestation runs in the background, the TOCTOU problem is limited to the interval in which each memory area is attested. This time span t_{hash} is very short due to the optimized implementation (see Section 3.7.1), so that manipulations of the flash memory can be detected. This also implies that no pseudorandom memory traversal is required, as used in SWATT [241].

Memory Manipulation. Another strategy to avoid detection is to use, create, or find unmonitored memory, for example, the RAM memory. The attacker could also try to change the memory layout by manipulating the partition table. However, in general, moving malicious code to RAM is not feasible. Code usually resides in flash memory, and RAM is therefore typically marked as non-executable [101]. Furthermore, manipulating the partition table is also a highly challenging task [102]. In the case of the ESP32, several preconditions must be met: first, SPI Dangerous Write must be enabled. Second, an entire flash page would need to be rewritten. Third, OTA (Over-the-air) updating typically implements partition changes and rewrites [100], which inevitably causes a reboot. Thus, manipulating the partition table is unfeasible for an attacker. Note that runtime reconfiguration of memory permissions is also not possible because the memory management unit (MMU) is privileged and set during boot.

3.7.3 Attacks on Attestation Protocol

There are two attack strategies to undermine the underlying security assumptions of the protocol used by REALSWATT:

1. Using multiple processor cores to accelerate the attestation function.

2. Optimization of the attestation loop accelerates the attestation process.

We discuss how REALSWATT addresses both techniques and ensures the security of the software-based attestation.

Using Multiple Cores to Break Attestation. The attacker can use the full computing power of the device, while REALSWATT has to obey strict limitations due to real-time critical jobs. The attacker could ignore real-time jobs and try to use more than a single core for hash calculations. In case hash calculation can be accelerated through parallelization, the attacker gains an attack window. However, the attestation process still cannot be accelerated using multiple processor cores since we use a Merkle–Damgård construction for hashing (such as SHA-1 or SHA-2). This method sequentially hashes each block and requires the previous block as input. Thus, an attacker cannot use multiple processor cores to parallelize this process. The Merkle–Damgård hash construction is strictly sequential.

Optimizing the Implementation of the Attestation Loop. Software-based attestation (SWATT) [241] is based on the assumption that an attacker cannot accelerate the implementation of the attestation algorithm. Thus, an optimal implementation of the attestation function and its main component, the hash function, is required. Algorithms are complex, and alternative ways exist to implement the same functionality. Castelluccia et al. demonstrated that SWATT [241] can be undermined using a faster implementation of the attestation function [57]. As described in Section 3.2, this gives an attacker a time slot where the attestation does not run, although the verifier assumes the prover is currently running the attestation function. RealSWATT addresses this issue using standard hash functions like SHA-256, for which optimized implementations or even hardware acceleration exist, as elaborated in Section 3.4.1. However, as these hash functions are widely used and have been well-studied in the past [193], it is unlikely that there exist implementations that significantly improve the execution speed, especially compared to hardware-assisted hash functions as we use in RealSWATT. Since we use global and standardized hashing algorithms (SHA-2) [193], highly optimized software implementations are available in case the target platform features no hardware hashing module.

3.7.4 Network-based Attacks

Connected devices are inherently prone to network-based attacks. Common practical issues in remote attestation, such as offloading the attestation process, have been considered in the design as discussed in Section 3.4. In the following, we explain which network-based attacks remote attestation faces and how these issues have been addressed in RealSWATT.

Shifting the Attestation to Another Device. One main problem in software-based attestation is offloading attacks. This means shifting the attestation task to an external entity. However, the IoT gateway prevents external communication. Hence, it is neither possible to leak attestation data (for example, the nonce) to a remote party nor to receive attestation reports from outside the network. Furthermore, all IoT devices in the network are covered by the attestation, so there are no free resources to perform an attestation on behalf of another device.

Delaying Communication. The attacker can delay the communication between the prover and the verifier. As some variation in the transmission time is normal, this would remain undetected to the verifier. While this would be a critical problem in traditional software-based attestation, RealSWATT attestation is designed to work with delays as imposed by communication, for example, using standard Wi-Fi networks. The concept of continuous attestation ensures that even if communication is delayed, the attestation process remains unaffected. Continuous attestation sends a second nonce before the previous attestation process has finished. The verifier times the second nonce such that it should arrive in time even under the worst expected network latency (see Section 3.4.1). All attestation runs on the prover are continuously running and have constant attestation time t_{att}. If the attacker intentionally delays communication, the verifier will detect the shift in communication delay, and the attestation time will exceed the threshold t_s.

3.8 Related Work: Attestation for Real-Time Systems

In the related work, we compare different approaches for remote attestation of embedded systems with real-time constraints. As discussed, such devices have special timing requirements that remote attestation schemes have to obey.

Remote attestation is often implemented as an atomic process that cannot be interrupted. Thus, this approach induces reduced responsiveness to external events such as interrupts. This is particularly difficult in real-time environments, as this conflicts with real-time constraints. All remote attestation architectures face this issue, including software-based [240, 241], hardware-based [3, 199] as well as hybrid [90, 91, 112, 144, 201, 202] architectures. So either the attestation process delays time-critical tasks, or the attestation process is being interrupted [200]. Both options are problematic.

Interrupting real-time tasks is not an option, as this is similar to a defect of the device. The other way round, a malicious or compromised prover could even hinder such a device from normal operation by regularly triggering attestation runs [50]. Attackers could exploit this behavior by regularly triggering events that interrupt the attestation, preventing any attestation run from completion. The attacker can also use this time to hide malware or cover the traces from a compromise.

Hence, remote attestation schemes for devices with real-time constraints have to consider the specific needs of these devices. There exist several attestation schemes that address these issues and fulfill the needs of devices with real-time constraints. One solution to this problem is to allow the attested device to schedule the attestation run by itself instead of invocation by the verifier. ERASMUS implements this approach by splitting the attestation phase into two separate parts. The prover performs the self-measurements for the attestation function according to a pre-established schedule. The verifier can collect these measurements later on. This takes advantage of the fact that the measurement itself requires a lot of computational power while transmitting the results to the verifier requires nearly no computations [55].

SMARM addresses the problem of the roving malware that hides itself by moving around on the attested device to evade detection by making the attestation process random and unpredictable [54]. So, the attestation function can be interrupted at any time. Even though the attacker can stop the attestation function, the attacker cannot predict which part of the device will be attested next. This way, malware cannot hide effectively from being detected by the attestation function by interrupting the attestation function and moving around the device.

TYTAN is a security platform for inexpensive embedded devices under real-time constraints [49]. It implements hardware-based code separation and communication between protected modules. It uses the FreeRTOS real-time operating system to ensure proper real-time scheduling. Furthermore, secure boot ensures the integrity of the code and data while loading. In contrast to the similar TrustLite [159] it allows dynamic reconfiguration of tasks and gives real-time guarantees. PAtt is an attestation scheme specifically designed to attest industrial control systems, considering the specific requirements of such devices. It does not require any custom hardware extensions or modifications and is designed to work on embedded devices that run fast control loops [121].

APEX is a hybrid remote attestation for embedded devices that allows proof of execution [203]. That is, APEX can guarantee that a specific functionality has been executed. It builds upon the VRASED architecture, which ensures static integrity [201], and the APEX attestation scheme [203], which also guarantees proof of execution. However, both VRASED and APEX are not suited for devices with real-time constraints [201, 203]. In contrast, the ASAP hybrid attestation scheme allows proof of execution of embedded real-time devices [58].

Real-time tasks are also problematic in hardware-based attestation schemes using a Trusted Execution Environment (TEE) such as TrustZone [24]. Switching into the secure environment, such as the secure world in TrustZone causes overhead and interrupts other processes [3]. One solution to implement secure hardware-based attestation is to integrate the attestation into the device's schedule. To prevent attackers from manipulating this schedule, moving the scheduling into a secure environment is also necessary. RT-TEE proposes a framework to run the scheduler of a real-time system inside a TrustZone TEE [282]. This approach has the advantage that attackers cannot maliciously manipulate the schedule.

Another solution is to implement the attestation functionality in hardware completely. This way, the normal execution is not impaired. Furthermore, changes to the attested software, such as instrumentation, are not required. However, this approach requires custom hardware. For instance, LO-FAT implements control flow attestation by extending the processor with a module that monitors the processor pipeline and monitors branches within the execution [85]. LiteHAX extends this approach. [84]. It can detect control flow as well as data flow attacks by monitoring both branch events and load/store instructions. In both approaches, attestation reports consist of a hash of the monitored execution. This hash is calculated in a hardware hashing module so that the normal operation of the attested device is not impaired.

3.9 Summary and Conclusions

In this chapter, we presented RealSWATT, a purely software-based remote attestation framework that allows to attest even systems with real-time constraints. RealSWATT is designed to work on legacy devices in real-world IoT scenarios. We achieve this by introducing continuous attestation, which constantly performs attestation in the background without interfering with the normal operation of the system by using a dedicated processor core. In the evaluation, we showed that RealSWATT attestation actually has predictable and constant runtime, a mandatory requirement for software-based attestation. We implemented RealSWATT into a syringe pump, a critical medical device with real-time requirements. In an end-to-end experiment, we successfully detected a compromise of the syringe pump via an insecure configuration interface. To show practicability, we integrated RealSWATT into ESPEasy, an open-source framework to use on commercial off-the-shelf IoT devices.

CHAPTER 4

User-Understandable Remote Attestation

Up to now, we focused on embedded devices with real-time constraints in corporate environments. But also in private households, there are many existing legacy devices. Therefore, in this chapter, we specifically target consumer IoT devices in home networks. The Internet of Things (IoT) enhances previously unconnected devices with Internet access. Popular examples are smart lamps, household appliances, security cameras, smart TVs, and smart speakers. Nowadays, IoT devices are ubiquitous: in 2019 the number of connected IoT devices already reached 35.7 billion [226]. IoT devices often perform important security-critical tasks, for example in a smart door lock, or operate in privacy-sensitive areas, such as a security camera or as a smart speaker [253].

Those IoT devices are worthwhile targets as typical IoT devices are black box systems to the user with a limited understanding of threats to security and privacy [299]. Especially from the perspective of end users, IoT devices behave like a black box: As long as they work as intended, users will not suspect or even detect any compromise. Typically, a user does not have any direct control over the software running on the IoT device, but solely configures and maintains IoT devices using a web interface or a companion app [12]. Hence, it is very likely that the user misses that illegal recordings and transmissions occur if a security camera or a smart speaker is hacked. Malware can be deployed before the device was purchased by the user or afterwards through backdoors and vulnerabilities. A compromise with malware will most likely not be detected if the functionality does not fail [11]. For instance, 48% of companies reported they are unable to detect whether an IoT device on their network suffers from a breach or is part of a botnet [119].

Remote attestation is a popular method to verify the integrity of a remote and untrusted device. Many attestation methods have been proposed for embedded and IoT devices [18, 54, 73, 87, 163, 164]. As discussed, the main challenge of remote attestation is to obtain trustworthy measurements from an untrusted device. However, many IoT devices neither feature trusted computing components for required for hardware-based attestation nor the necessary hardware extensions for hybrid attestation. Replacing or extending these legacy IoT devices to enable hardware-based or hybrid attestation protocols is often not an option in practice; especially considering that the computing hardware is deeply integrated into the respective IoT device. Consequently, software-based attestation is often the only viable approach.

For hardware-based and hybrid attestation schemes, the attested devices feature secret keys to authenticate themselves. However, as there is no secure key storage in software-based attestation, there is no root of trust that allows authentication of the attested device. In fact, any information on the attested device is accessible to the attacker. Hence, an attacker can relay attestation requests to a different or even a simulated device without being noticed by a remote verifier. Such so-called offloading attacks, that is, forged attestation reports from other devices, are an inherent and one of the most significant problems in software-based attestation due to the lack of a secure root of trust.

To address this lack of device identification due to the missing root of trust, we proposed for the RealSWATT attestation framework in Chapter 3 to use an IoT gateway to limit the communication with other, unattested devices and detect so-called offloading attacks. While this approach is adequate for usage in industry and other centrally managed environments like hospitals and companies, this is infeasible for private and home setups. In these environments, there are often many devices, of which many are vulnerable [12]. Assuming that users have a secure IoT home setup and are continuously monitoring all Internet traffic is unrealistic for home users.

Contributions. In this chapter, we propose SCAtt-man, the first remote attestation scheme specifically designed to allow user-observable attestation, thereby solving the problem of missing device authentication in software-based attestation. For the first time, we exploit side-channel information such as light or sound that are observable by the user to evaluate the attestation result allowing the user to identify the device that is being attested and detect offloading attacks.

Although it has been popular to use side-channels such as sound [305], ambient sound [129, 179, 236], or acceleration [134, 289] for context-based authentication, that is, key exchange or device pairing, we are not aware of any remote attestation scheme that leverages such side-channels. We give an overview of context-based authentication schemes in Section 4.8. Using such aforementioned communication side-channels for attestation is not straightforward, as implementing a secure software-based attestation scheme involves tackling manifold challenges, as we elaborated in Chapter 2.

The usage of side-channels allows secure deployment of software-based attestation on legacy IoT systems without unrealistic hardware requirements and changes. In fact, we demonstrate that users can use their smartphones together with built-in sensors and actuators to attest the IoT device.

Attestation via side-channels has the following advantages:

1. Communication is user-observable.
2. Communication is limited to short distances, limiting a remote attacker.
3. The Internet connectivity of the attested device can be interrupted to prevent offloading attacks.
4. The transmission time can be predicted precisely, which is a crucial requirement for software-based attestation.

The missing device authentication in software-based attestation is being replaced by the user, who can manually identify the device that is currently being attested. This approach offers users an intuitive way to identify IoT devices and the devices' integrity. It makes SCAtt-man the first framework for user-friendly software-based attestation. SCAtt-man does not require complex profiling or measurements during installation. If the users' smartphone already knows the correct configuration of the attested device, for example because it was used to initially configure the device, the smartphone can confirm the correctness of the attested device's configuration, the integrity of the device's software, and detect unwanted modifications or malware. In practice, this attestation functionality can also be integrated into the vendor's companion app that is often used to configure or use the IoT device [12].

In summary, we provide the following contributions:

- We propose SCAtt-man, a new attestation scheme that leverages side-channels for software-based attestation that the user can observe, thereby efficiently solving the root of trust problem in software-based attestation.
- SCAtt-man works on legacy hardware avoiding the need for additional hardware modules or actuators. Instead, it uses built-in hardware features, such as the built-in microphone and speaker of a smart speaker and a standard smartphone as a verifier.
- We implemented SCAtt-man in a smart speaker based on the popular ESP32 microcontroller and implemented a verifier as an Android app.
- We implement an audio communication protocol which is used for the attestation. In a measurement study, we show the reliability of the communication using this protocol. Furthermore, we profile the runtime of the attestation function, crucial to software-based attestation.

- In the evaluation, we show, based on a full end-to-end example, how SCAtt-man detects a real-world attack on a smart speaker via an insecure configuration interface (Section 4.6.4). In extensive experiments, we verified that SCAtt-man ran without any false positives and negatives, allowing a reliable attestation.

- We performed a user study to evaluate the usability and user experience of SCAtt-man (Section 4.6.5). Our user study not only showed that SCAtt-man provides good usability, but participants also stated that they actually believe that attestation can detect a device's compromise and that they would use this functionality if their own devices featured such an attestation functionality.

4.1 Background: Smart Speakers

In this section, we explain the foundations and concepts required to understand SCAtt-man.

Smart speakers are IoT devices that take natural spoken language as input and react appropriately by responding using synthesized voices or performing tasks. These tasks range from the supply of information such as the current time or weather, telling jokes, playing music, over the setting of timers and creation of lists, up to sending messages and controlling other connected IoT devices such as lights or smart locks [138]. Smart speakers are very popular; In 2020, more than 150 million units have been sold worldwide [51]. Studies have shown that users are especially concerned about the security of smart speakers. Although users often do not see threats in devices that do not record audio or video, such as smart plugs or light bulbs, they are aware of the privacy risks of smart speakers and IP cameras [299, 306]. Smart speakers feature microphones, audio-processing hardware, and speakers and constantly listen for a so-called 'wake word' [288]. This wake word starts the interaction with the smart speaker: the device then records the user's voice and sends it to a cloud service for further processing [89].

Smart speakers suffer from a large attack surface as they incorporate a complex architecture [89]. That is, they combine an IoT device with a combination of local and cloud features, including natural language processing techniques. The functionality of smart speakers can often be enhanced with third-party extensions developed by external vendors. For example, there are more than 18,000 English extensions available for Google Assistant [158]. For Amazon's Alexa speech assistant, more than 80,000 so-called skills are available in the US [157]. Malicious extensions can illegally access sensitive user data [67, 131, 264, 304], eavesdrop on private conversations [131], or even take over the smart speaker and connected smart home devices, including a door lock and the home security system [186].

4.2 Challenges

Remote attestation allows the detection of attacks on remote systems, such as malware and other software attacks, even on compromised devices. Therefore, attestation is capable of enhancing trust in connected systems. However, obtaining reliable self-measurements on untrusted and potentially compromised devices involves tackling several challenges. It is especially challenging to secure devices that lack hardware security modules like trusted computing components, for example, a trusted platform module (TPM) or a trusted execution environment (TEE). However, such components are often unavailable on IoT devices due to cost reasons. Enhancing legacy devices with trusted computing techniques is not an option due to the nature of embedded and IoT devices. The hardware is specially adapted to its needs and deeply integrated into the device so that the hardware cannot be replaced. This is a general challenge for embedded devices, but is especially true for consumer IoT devices, where a hardware replacement would rather involve a complete replacement of the device. Hence, software-based attestation is the only possibility to allow remote attestation of legacy IoT devices.

IoT devices are often integrated into complex ecosystems, where multiple devices are working together, communicating over the Internet, and relying on the vendor's cloud services. This poses challenges towards timing measurements and reliably identifying devices in software-based attestation, that is, precise timing measurements require uninterrupted and direct communication. An unexpected distortion in communication, for example, an unexpectedly long delay will make the attestation fail [241]. Often, software-based remote attestation even requires one-hop communication [255]. The lack of reliable device authentication allows attackers to shift attestation tasks to other devices or emulate devices without being detected. To solve this problem, communication with other devices has to be restricted. For example, to allow software-based attestation for IoT devices, the usage of a dedicated IoT gateway has been proposed to prevent the attested device to communicate with other, unattested devices, as laid out in Chapter 3. This approach relies on successfully detecting all offloading attacks, in which the attested device relays attestation requests to an external party over the Internet, and on all devices in the local network being benign to prevent collusion attacks. While this is a solution for managed environments such as companies, this is not a practical solution for home deployments. In home environments, many devices are not centrally managed. This includes both IoT devices which are often vulnerable [12], as well as more powerful devices like PCs and smartphones.

Besides these technical challenges, for actual, practical usage by end users also other aspects come into place. For many users, IoT devices are black box systems. The computer systems are deeply integrated into these IoT devices, users are supposed to use these devices as-is, not modifying or changing the software of these devices.

While this circumstance eases the usage of such devices, this also limits the user's understanding of the actual functionality of such devices. Designing systems so that the user can actually verify a device's integrity, and hence put trust into this particular device, is a difficult task combining technical soundness and usability.

Based on the general challenges for secure remote attestation that we described in Chapter 2.5, we derive the requirements for the software-based attestation of IoT devices, considering the specific circumstances in home environments. Implementing such a secure attestation scheme for IoT devices poses the following challenges to the design and implementation:

Challenge 1: Secure Self-Measurement. Secure software-based attestation requires careful implementation. The security of software-based attestation relies on its deterministic minimal runtime. As discussed in Section 2.7.2, an attacker may not be able to speed up the execution. If an attacker can find a faster implementation of the attestation function, the saved time can be used to compromise the attested device or alter attestation reports.

Challenge 2: Authenticity and Offloading Attacks. In software-based attestation, attested devices do not feature a hardware root of trust, but the security solely relies on timing properties. Hence, the verifier cannot securely authenticate the attested device and detect if an attested device is replaced by a different device or a simulation, or if an attestation request is relayed to another instance.

Challenge 3: Precise Response Times. IoT devices often use indirect communication through cloud services. They do not operate on their own but are integrated into ecosystems consisting of multiple different devices and are closely operating with their vendor's cloud services. Indirect communication is susceptible to relay attacks and increases round trip times and makes response times more fluctuating, complicating software-based attestation. Recall that precise prediction of response times is crucial for software-based attestation, as discussed in Section 2.7.2.

Challenge 4: Usability. For the user, IoT systems behave like a black box. Cryptographic functions, Internet communication, and attestation protocols, in particular, are abstract and unintuitive concepts. Therefore, developing user-understandable attestation protocols that users intuitively understand is crucial to gain the user's trust in IoT devices, especially in critical security and privacy domains.

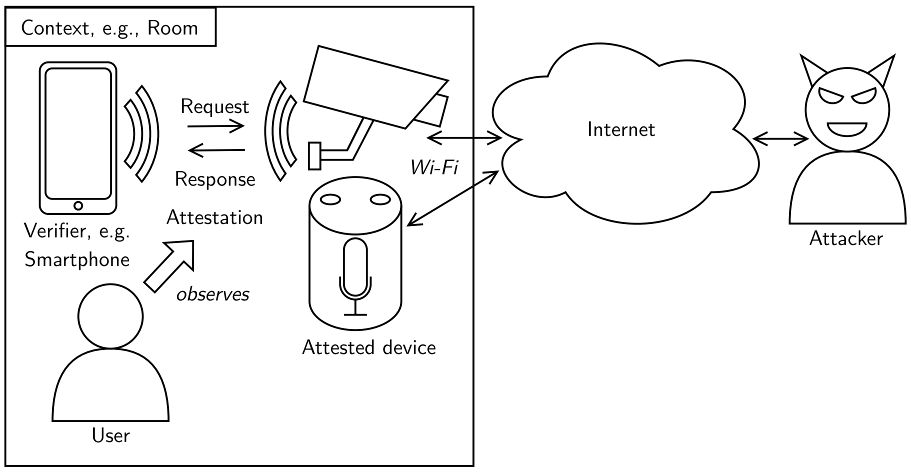


Figure 4.1: Threat model of SCAtt-man. The attested device and the user's smartphone are in the same context, in this case in one room. The attacker is outside the room and has no access to the context.

Challenge 5: Legacy Devices. There are billions of legacy IoT devices that do not feature trusted computing components. Security solutions for these systems must not require additional hardware extensions, extra sensors, or new communication technologies.

SCAtt-man addresses these challenges by developing a user-observable communication channel to perform software-based attestation. As we will show, this effectively solves the root of trust problem in software-based remote attestation and results in a user-comprehensible attestation process.

4.3 Assumptions and Threat Model

Figure 4.1 shows the general threat model of SCAtt-man. The verifier and the attested device share the same context, for example, because they are in the same room. The user can observe both devices. We assume a remote attacker outside of the context, typically connected via the Internet. This is a standard scenario for consumer IoT devices such as smart speakers or IP cameras.

4.3.1 Assumptions

We consider an untrusted IoT device that features a suitable sensor and actuator to be used as a side-channel for communication. The following relates to the implementation of SCAtt-man on a smart speaker. In this case, we leverage a microphone and a speaker, as they are always available in smart speakers. Note that other combinations are also possible, for example, a light sensor or camera and a lamp or display [234]. Furthermore, we assume that the user is close to the IoT device and within reach of this side-channel. The user has a trusted device for verification with suitable sensors and actuators. In our implementation, we use a standard a smartphone with built-in speaker and microphone.

We furthermore assume that there are no ambient disturbances. In particular, we assume that the attacker is not in range of the sensors, that is, we consider a remote attacker. Finally, we assume that the Internet connectivity of the attested device can be interrupted, for example, by using a switch. Alternatively, we also propose a solution to limit the Internet connectivity of the attested device using the user's smartphone. To do so, both the smartphone and the IoT device feature a Wi-Fi interface.

4.3.2 Threat Model

The threat model is depicted in Figure 4.1. We assume that a remote attacker can remotely compromise the smart speaker and for example, alter the configuration or install malware. This can be achieved by the exploitation of typical software vulnerabilities, such as memory errors [12], or insecure or insufficiently protected interfaces, a common problem in IoT devices [208]. Similar to all existing software-based attestation proposals, we do not consider physical attacks [241, 254]. This means the attacker cannot alter the hardware of the attested device, and for instance, replace or modify it. Furthermore, the attacker is not in range of the sensors used for the communication side-channel, that is, in the hearing range. We assume the user's smartphone to be trusted. This is a typical scenario for IoT devices, where the attacker is located outside the home and hence has no direct access to the device.

4.4 Concept of SCAtt-man

The goal of SCAtt-man is to make communication observable by the user, allowing the user to oversee the attestation process and identify the device being attested, thereby solving the inherent problem of missing authentication in software-based remote attestation. We achieve this goal by using a side-channel for the communication between the verifier and the prover. While in traditional attestation schemes communication is assumed to take place via wires or wireless, such as Ethernet or Wi-Fi, we explicitly opt for alternative means of communication. The limited reach of such transmissions reduces possible attacks. Furthermore, in contrast to radio communication, attacks will be noticed by the user.

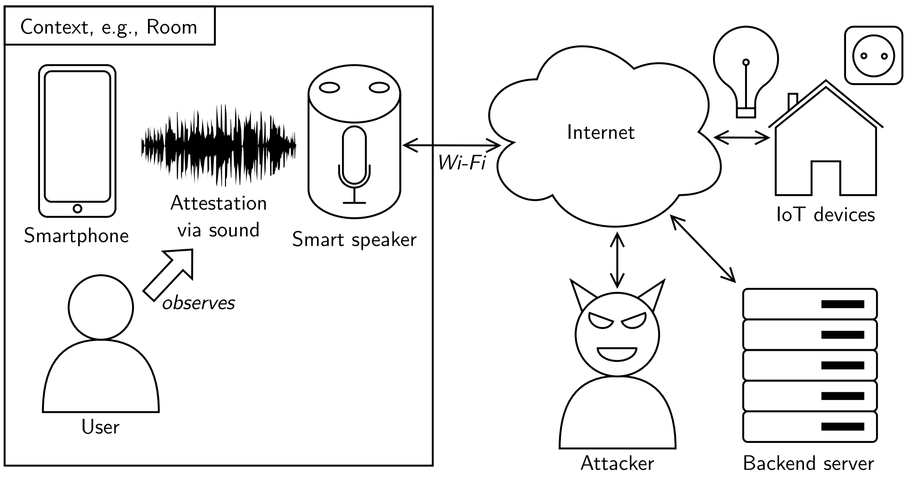


Figure 4.2: Concept of SCAtt-man attestation: The user can observe the communication between the prover and the verifier.

A suitable communication channel fulfills two properties:

1. It is user-observable and
2. It can be sent and received with built-in sensors in both the attested device and the device used for verification, such as a standard smartphone.

For example, well-suited communication channels are sound and light. A standard smartphone features a microphone and speaker, as well as a camera and a display or a LED (Light-emitting diode) used as a flash.

In this section, we explain how we addressed the challenges described in Section 4.2 and developed a proof-of-concept of SCAtt-man into a smart speaker. Figure 4.2 illustrates the concept of a smart speaker enhanced with SCAtt-man attestation. As explained in Section 4.1, smart speakers are a good example to show the applicability of SCAtt-man attestation. First, smart speakers are a typical and popular type of IoT device. Second, users are particularly concerned about the security of those devices [299, 306]. This is an important aspect, as the user needs to initiate the SCAtt-man attestation manually.

The communication between the prover, that is the the attested device, such as a smart speaker, and the verifier, in this case the user's smartphone, is performed via audible sound. This makes the attestation process user-observable. The communication consists of digital data encoded into sounds. The user doesn't need to be able to decode the data, it is sufficient that the user is able to identify the devices that are communicating. This way, offloading attacks, that is, when another device responds to the attestation request, are effectively prevented. Furthermore, a remote attacker is not able to influence this local communication.

Due to this communication channel, the attestation cannot be run in the background but requires interaction with the user. Therefore, users manually run the attestation process. To do so, we provide a smartphone app that guides users through the attestation process. The app explains the necessary steps, runs the attestation, and shows the result. We explain the functionality of the app in Section 4.5.2 and show details of the attestation process in Section 4.5.5.

Although using side-channels for attestation seems like a straightforward concept, the development of a secure software-based attestation scheme using side-channels needs tackling specific challenges such as developing a suitable communication protocol and coping with the manifold attacks on software-based attestation. In the following, we explain how we developed a reliable audio transmission protocol, a secure attestation function, and restrict Internet access for the attested device to prevent offloading attacks.

4.4.1 Audio Protocol

Data-over-sound describes the concept of sending digital data via sound waves. Sending data over sound is a well-established concept. For example, it was used in acoustic coupling to connect computers via the telephone network [287]. To send data over sound, the sending device encodes the data into sound patterns. These are then sent by a speaker and received by a microphone [195].

For SCAtt-man, we need a transmission protocol that is user-observable meaning that it works within frequencies of standard microphones and speakers of smartphones and smart speakers and is within the human hearing range. A strict requirement for software-based attestation are predictable execution and transmission times. Hence, the protocol should feature a fixed message length and do not have error-correction codes such as parity bits. Otherwise, an attacker could exploit this circumstance to gain a timing advantage by starting the attestation process before the transmission has been completed by using the error correction to complete incomplete transmissions.

So, on the one hand, reliable audio communication between the attested device and the verifier is crucial for a secure attestation: The user cannot determine whether a communication error or a compromised device causes a failed attestation run. On the other hand, the audio protocol may not allow speedup by omitting parts of the transmission such as checksums or parity bits to allow an attacker to accelerate the communication. As the security of software-based attestation relies on a predictable execution time, this restricts the usage of error corrections. For example, the original SoniTalk protocol [300] repeats after each transmission the inverse data. As this redundant information is not strictly necessary in most cases, an attacker could already start the attestation run before the audio transmission is complete, leaving a gap in which the smart speaker could be compromised. Therefore, we removed this feature to ensure that the audio communication cannot be accelerated, Data-over-sound protocols have many parameters, for example, frequencies, encoding of bits, and duration of the transmission. We perform a detailed explanation of the implementation of the data-over-sound protocol in Section 4.5.3. In Section 4.6 we extensively tested and optimized the data-over-sound communication by fine-tuning these transmission parameters to ensure a reliable attestation.

4.4.2 Attestation Function

The main component of any attestation scheme is the attestation function. The attestation function takes the self-measurement of the attested device which is then transmitted to the verifier. However, as elaborated in Section 4.2, performing a trustworthy self-measurement on an untrusted device without specialized hardware is a challenging task.

SCAtt-man attests the integrity of software and configuration of the attested device. However, we restrain SCAtt-man to static attestation due to the infrequent attestation runs: The user must manually initiate every attestation run. Furthermore, the attestation cannot be performed in the background as it plays audio and fully utilizes computing resources of the attested device, forcing one to pause the usage of the speech assistant while the attestation is executed. More sophisticated attestation schemes like runtime such as control flow attestation [3] and data flow attestation [4, 84] need to be run in the background during normal operation of the attested device, and rely on frequent communication with the verifier during attestation. Running attestation without sending the results to the verifier does not give any security benefit as a compromise will not be detected.

There are several aspects to be considered when designing an attestation function:

Optimal Implementation. The security of software-based attestation solely relies upon the computational capabilities of the attested device and timing threshold, that is, measurements of the execution time of the attestation function. This induces that an attacker cannot significantly accelerate the execution speed of the attestation function. We solve this problem by using built-in hardware modules to run the attestation function. The hardware-accelerated execution of the attestation function is faster than any software-based implementation on the same device. In case no hardware-based acceleration is available, we use standard and widely-used hashing functions. To prevent acceleration using parallelization, we chose a hashing function that use the Merkle–Damgård scheme, which does not allow parallelization [182].

Replay Attacks. In replay attacks, the attacker responds to an attestation request with pre-computed or old attestation reports. SCAtt-man prevents such attacks by including a random nonce into the attestation request. This nonce, chosen by the verifier and hence out of control of the attacker, ensures freshness of the attestation reports.

Empty Memory. An attacker can use any memory not covered by the attestation. Therefore, it must be ensured that the attestation function actually covers all executable memory and that the attacker cannot compress any memory to obtain unattested memory which can be used to store malicious code. SCATT-MAN addresses this problem by closely monitoring execution times of the attestation function. Deviations of the runtime of the attestation functions due to the on-the-fly decompression of data, while the attestation is running in parallel, will be detected.

Runtime. Determining the correct runtime of the attestation function is crucial for the security of software-based attestation. Therefore, we designed the SCATT-MAN attestation function such that its runtime can be configured by increasing the number of iterations. This feature can be used to obtain a runtime that can be clearly distinguished from compromised ones within the attestation process. In Section 4.6.1, we investigate in detail the runtime of the attestation function to determine strict thresholds and distinguish between correct and compromised runs of the attestation function.

The attestation function of SCATT-MAN uses a hash function to obtain a measurement of all software and configuration data on the attested device. For example, a device can have multiple partitions where executable code and data are stored. We hash all of these memory areas. More information about implementing the attestation function can be found in Section 4.5.2.

4.4.3 Limitation of the Internet Access

Preventing offloading to an external party is crucial for secure software-based attestation. Offloading means relaying the attestation request sent by the verifier to another device. Because there is no physical security in software-based attestation, there is no way to authenticate the device being attested besides using response times. Furthermore, the runtime of the SCATT-MAN attestation function on the attested device is longer than transmission to an external party, like a cloud service or another IoT device takes. Thus, external transmission must be prevented. There are two possibilities to achieve this. First, use a hardware-based method that disables communication like a hardware kill switch. In case of a wired connection, it would also be sufficient to unplug the network connection. Alternatively, we propose a software-based solution to limit the Internet connection via Wi-Fi of the attested

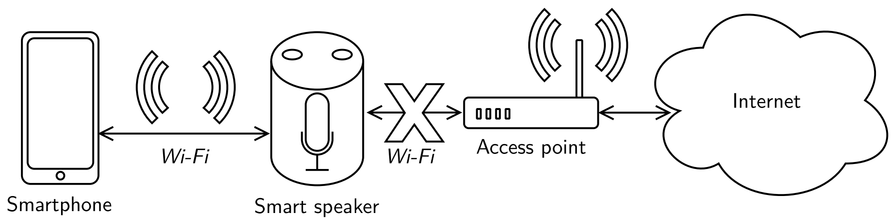


Figure 4.3: Limiting Internet access in SCATT-MAN attestation: The attested device and the verifier initiate a direct Wi-Fi connection, so that the attested device cannot connect to any other network, thereby preventing an Internet connection.

device. This is particularly suitable for legacy devices that do not feature alternative methods to deactivate the Wi-Fi connection.

Hardware Kill Switch. Integrating a hardware kill switch to deactivate network functionality is an elegant solution as it is a simple, user-understandable concept. Furthermore, the usage of such a button is not limited to SCATT-MAN attestation. For instance, there are smartphones for privacy-aware users that feature a hardware kill switch to deactivate Wi-Fi and radio communication [80, 216, 220]. Keep in mind that research found that even encrypted traffic of IoT devices can be used to monitor actions [6]. As discussed earlier, users are privacy-sensitive about devices that process speech and pictures, therefore such a button would be beneficial to give users control over the device. Many smart speakers already feature a button to mute the microphone [170].

Software-Based Locking. Alternatively, for the case that there is no possibility to physically prevent an Internet connectivity of the attested device, we also provide a software-based solution. It does not have any hardware requirements, but works solely in software. We achieve this by configuring the attested device as a Wi-Fi access point. The smartphone then connects to this access point and continuously checks the availability of the attested device to ensure that the attested device cannot connect to a third party. This prevents the attested device from connecting to the original Wi-Fi network, and thereby the access to the Internet. Figure 4.3 shows this case. However, some Wi-Fi chips can keep connections to multiple simultaneous connections. For example, the ESP32 supports a combined 'station and access point' mode, which allows the ESP32 to open a Wi-Fi access point while being connected to another Wi-Fi network [105, 197]. However, this is limited to the same channel as the radio can only listen to a single channel. Consequently, if the verifier maintains a continuous connection to the attested device on a channel different from the regular Wi-Fi connection. Hence, the attested device cannot connect to any other Wi-Fi network to obtain a connection to the Internet. Thus, Internet access of the attested device is effectively prohibited.

Summing up, the nonces that initiate the attestation as well as the attestation reports are transmitted via audio between the attested device and the verifier. During attestation, the attested device is disconnected from the Internet to prevent offloading attacks by prohibiting communication with the attacker. The attestation function is designed such that the execution cannot be accelerated without altering the hardware. In Section 4.5 we show how we combined these techniques into a secure attestation scheme and implemented them into a smart speaker.

4.4.4 Attestation without Human Interaction

SCAtt-man was primarily designed to address attestation in IoT home installations allowing end users to verify the integrity and trustworthiness of their devices. By design, SCAtt-man requires the user to start and observe the attestation process. But SCAtt-man attestation can also be adapted to work without human interaction.

To do so, two things have to be considered. First, the manual steps need to be automatized, and second the verifier app that currently runs on the user's smartphone needs to be replaced. For example, the button to trigger the attestation could be replaced by a timer automatically starting the attestation. Furthermore, the attestation could be performed either by a dedicated trusted attestation device or another nearby smart speaker. This way, these devices could implement a mutual attestation protocol using our proposed sound side-channel.

4.5 Implementation

In this section, we describe the implementation of the key components of SCAtt-man. Figure 4.4 gives an overview of our implementation. While SCAtt-man is also suitable for legacy devices, as it does not require special hardware, the implementation on state-of-the-art smart speakers such as Amazon Echo is not easily feasible. The hardware of these devices is locked down, hindering the replacement of the software. Furthermore, the software of these devices is closed-source, so that modifications are not easily possible. Therefore, we developed a custom smart speaker enhanced with SCAtt-man attestation, basing on publicly available web services and hardware that is commonly used in IoT devices. The implemented attestation scheme allows to verify both the program code as well as configuration data. For the verifier, we built an Android application that implements the verifier logic and guides the user through the attestation process. In the following, we explain the attestation functionality and the data-over-sound protocol in detail. Furthermore, we show how

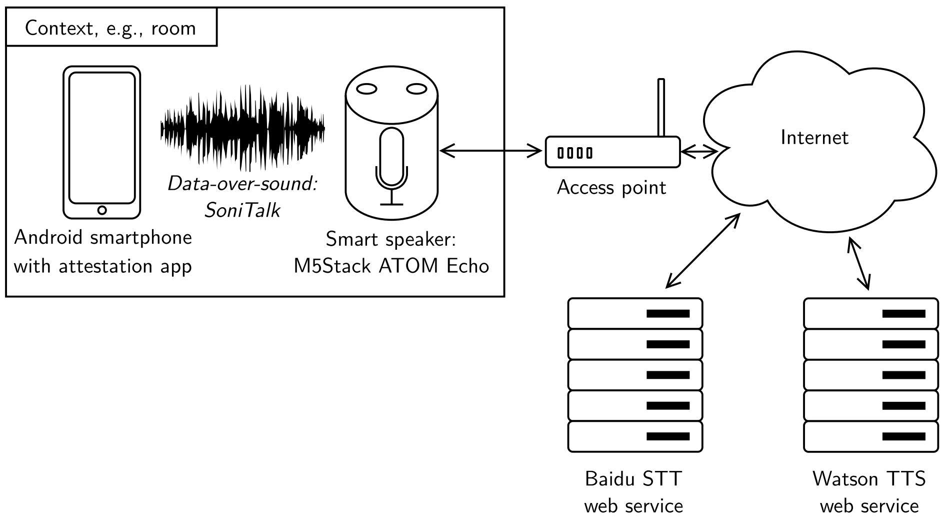

Context, e.g., room
Data-over-sound: SoniTalk
Android smartphone with attestation app
Smart speaker: M5Stack ATOM Echo
Access point
Internet
Baidu STT web service
Watson TTS web service

Figure 4.4: Implementation of a smart speaker with SCAtt-man attestation.

the user experiences the attestation process. In Section 4.6, we verify the attestation functionality and show SCAtt-man can detect compromises. Furthermore, we evaluate the reliability of the data-over-sound transmissions.

4.5.1 Smart Speaker

To implement the smart speaker, we used an M5Stack ATOM Echo module[1] shown in Figure 4.5, that combines the popular ESP32 microcontroller with an integrated microphone, a speaker, and a configurable RGB status LED as well as a button. The ESP32 microcontroller is deployed in various IoT devices [96], it features a dual-core processor, and a Wi-Fi module [103]. On this platform, using the popular FreeRTOS [116], we integrated basic smart speaker functionality, that is, recording voice commands, sending them to a cloud service, as well as receiving and playing back the response via the integrated speaker. Note that the usage of FreeRTOS or other operating systems is not mandatory to integrate SCAtt-man, as SCAtt-man does not require complex process structures or scheduling. When the button is pressed, the smart speaker records the voice command and directly streams the voice command via an HTTP connection to a speech-to-text (STT) cloud service[2]. This service determines the spoken text from the recorded sound and sends it back to the smart speaker. The smart speaker then processes the command. Using the text-to-speech (TTS) functionality of the IBM Watson REST (Representational

[1]`https://docs.m5stack.com/en/atom/atomecho`
[2]`https://fanyi-api.baidu.com/`

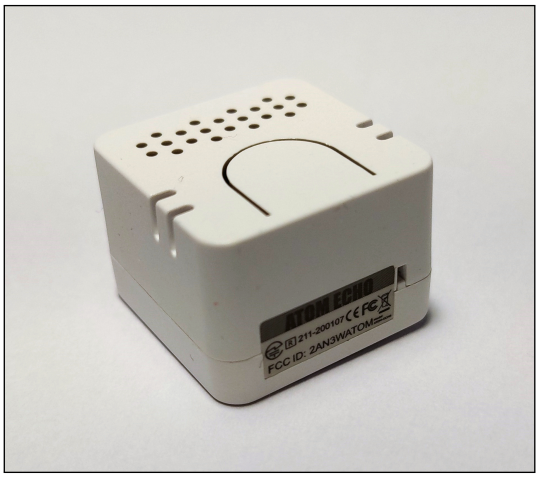


Figure 4.5: The M5Stack ATOM Echo.

state transfer) API[3], the voice assistant can convert any text to spoken word. The speech assistant sends a string and receives wave audio, which is played back later. This is a similar operating mode as for standard commercial voice assistants. More information on smart speakers in general can be found in Section 4.1. The current status of the smart speaker is indicated by the color of the status LED, allowing the user to always check the current execution mode of the smart speaker, for example, recording, speaking, or the current step within the attestation process.

4.5.2 Attestation Process

With a long button press, the smart speaker switches to attestation mode, which is confirmed by a red LED light. In attestation mode, the ESP32 microcontroller switches from Wi-Fi station mode to Wi-Fi access point mode. The verifier application then connects to the access point of the ESP32. However, the connection of the smartphone on a different radio channel, other than the home Wi-Fi network, disables the ESP32's capability to maintain an Internet connection during attestation. Once the smartphone connects to the access point, a sound listener is started, which receives the nonce from the smartphone. The received nonce is then passed to the attestation process, which computes the attestation report. The attestation report is then transmitted back to the smartphone. Lastly, the ESP32 switches back to the Wi-Fi station mode and connects to the Internet to resume normal operation.

For the attestation process, we suspend all tasks not related to the attestation using the FreeRTOS API (*vTaskSuspend*) [115]. Thus, we ensure that all resources are

[3]`https://www.ibm.com/cloud/watson-speech-to-text`

available for the attestation. Furthermore, we use a hardware-accelerated SHA-256 for hashing. SHA-256 is a Merkle–Damgård [182] construction [214], that follows a strictly sequential process: Each hash block is used as input for the subsequent block. Therefore, the hashing process cannot be parallelized. Our attestation includes all code and data sections of the ESP32. This is because the ESP32 organizes its code and data sections as partitions, which are all covered by the attestation function. The received nonce initializes the hashing. Since the RAM of the ESP32 cannot fit the entire code and data space, we split the code data and partitions into blocks of 256 byte each. Other block sizes are possible but will lead to different attestation runtimes. The choice of block size has no impact on the integrity of the attestation itself. We use the hash of each block as input for the next one, thus different block sizes will yield different hashes. We loop the hashing of the entire memory of the ESP32 to achieve a suitable attestation runtime. Finally, the resulting hash is transmitted to the smartphone, and we resume all suspended tasks.

4.5.3 Data-Over-Sound

As already explained in Section 4.4, we use a short-range side-channel for the attestation process. This allows the user in proximity to the IoT device, in our case the smart speaker, to perform remote attestation without the risk of a remote attacker hijacking the communication channel. In fact, the attestation can be performed on an IoT device with network connections completely turned off. For example, such a short-range side-channel would be sound. In addition, this side-channel is perceptible by users. Therefore, only devices in short physical proximity (for example, the same room) can interfere with the communication. In order to transmit data between the smartphone application and the smart speaker, we implemented a data-over-sound protocol based on SoniTalk [300, 301]. We introduced the following changes to the SoniTalk protocol to adapt it to the requirements for software-based attestation. (1) We introduced a fixed message length, (2) reduced the number of frequencies to increase reliability, and (3) removed the transmission of inverted message blocks.

We have chosen the fixed message length of 32 bit, as it corresponds to the chosen length of the transmitted data (that is, nonces and hashes). A single message is split into eight blocks ($m = 8$) each with a length of 4 bit ($n = 4$) each. Those n bits are encoded by the presence of a corresponding carrier frequency. We reduced the number of carrier frequencies to four since we implemented the entire attestation process on a low-end device (M5Stack ATOM Echo) with low quality sensors (speaker and microphone). A reduced number of carrier frequencies makes the transmission process more robust. This is crucial for software-based attestation, as the user cannot distinguish between a failed attestation due to a transmission error and a real compromise. In order to avoid attestation ahead of time, we completely removed the inverted message blocks. Hence, an attacker is required to wait until the data transmission is finished. As our evaluation in Section 4.6 shows, our chosen protocol parameters allow reliable communication such that the redundant information (that is, inverted message blocks) is not needed. Each message consists of one start block M_S, followed by m message blocks M_m. Through an empirical study as shown in Section 4.6.2, we set the transmission time of a message block to 240 ms. Thus, the transmission of a message takes $9 \cdot 240\,\text{ms} = 2160\,\text{ms}$.

Our protocol's generic data transmission process is visualized in Figure 4.6. We use four carrier frequencies. The presence of a carrier frequency indicates a bit value of 1, and the absence of a frequency indicates a bit value of 0. The message blocks are transmitted in sequence. The message block is always present for the time span of $d = 240\,\text{ms}$. Figure 4.6 shows also an optional pause p between message blocks, which has been set to zero in our implementation.

Data-Over-Sound on the Smartphone We have implemented the verifier as a user-friendly Android application. Since a smartphone has a good quality microphone and speaker, it is well suited to communicate with the IoT device over the sound side-channel. Furthermore, it offers a familiar and well-known interface for the user. The verifier application implements a sending and receiving module according to the used data-over-sound protocol (see Section 4.5.3). The sending module has two main components: *Encoder* and *ToneGenerator*.

The *Encoder* is responsible for splitting the message (fixed length of 32 bit) into message blocks (each 4 bit). The *ToneGenerator* generates the sounds for each message block. The sound is generated by resampling the active frequencies. The resampling is conducted using a high-resolution sinus lookup table. Afterwards, the active frequencies are stacked by adding the sampled values. In order to avoid clipping in the audio playback, the stacked tone is normalized to a common gain. Once all tones are sampled and normalized, the message blocks are played sequentially.

Receiving takes a little more effort, therefore there are more components in the Android application: *Recorder*, *AudioCalculator*, *FrequencyCalculator*, *RealDoubleFFT* and *Decoder*. We based these components on the Android Audio Sample Project[4]. The *Recorder* component records and stores sound. The application needs about 60 ms to record, which yields four samples per interval. On the sending side, each tone is played for 240 ms. After recording the audio samples, we perform a Fast Fourier Transform (FFT) with the three components *AudioCalculator*, *FrequencyCalculator* and *RealDoubleFFT*. The Fast Fourier Transform (FFT) yields the frequencies from which the sound is composed. After the frequency decomposition, the *Decoder* can determine the start block and convert each message block to 4 bit integers. In order to properly receive messages over sound, the recording of message blocks must be

[4] `https://github.com/lucns/Android-Audio-Sample`

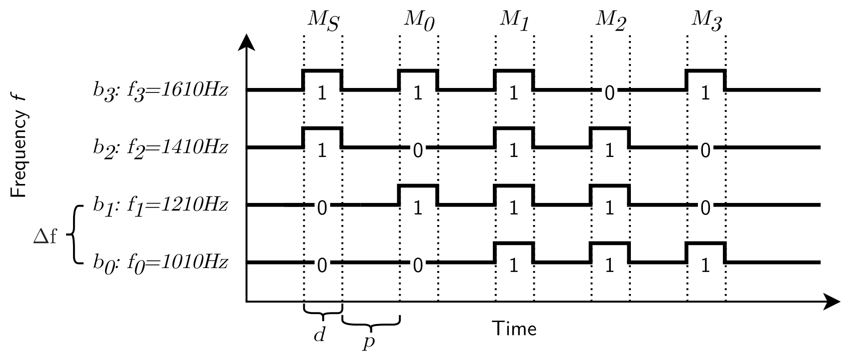


Figure 4.6: Example of a data-over-sound transmission: Transmitting 01011111 11101001 after starting with the M_s signal.

synchronized with the playback of the message. Thus, the application waits until a specific volume threshold is exceeded and begins recording the message. If the start block M_S is not received, the application waits until the next time the volume threshold is exceeded.

Data-Over-Sound on the ESP32 In contrast to integrating the data-over-sound protocol on a smartphone, implementing a resource-constraint device such as the ESP32 is more challenging. Typical commercial smart speakers like Alexa[5], Google Nest[6], or Apple Homepod[7] offer more specialized audio hardware and resources. However, if we implement data-over-sound on the ESP32, which is a generic and popular IoT device, we show the feasibility of our side-channel attestation scheme for a broader range of devices. Sending data via the data-over-sound protocol is straightforward as the ESP32 can encode data on-the-fly. This process can be implemented similarly to the sending process on the smartphone application. However, the receiving component on the ESP32 is challenging due to resource constraints, especially memory constraints on the device. For example, recording and storing large sound blocks is infeasible. Furthermore, the ESP32 provides only a limited number of hardware-accelerated Fast Fourier Transform (FFT) functions. We solved this challenge by analyzing only the subframes (fraction of 32 ms) of each transmitted tone. We also limited the number of frequencies and separated those frequencies by at least 100 Hz. Therefore, the FTT can detect the active frequencies in each tone more easily. Those optimizations to the SoniTalk protocol enable reliable data transmission over audio on resource-constraint devices such as the ESP32.

[5]`https://www.amazon.com/smart-home-devices/b?node=9818047011`
[6]`https://store.google.com/product/nest_audio`
[7]`https://www.apple.com/de/homepod-mini/`

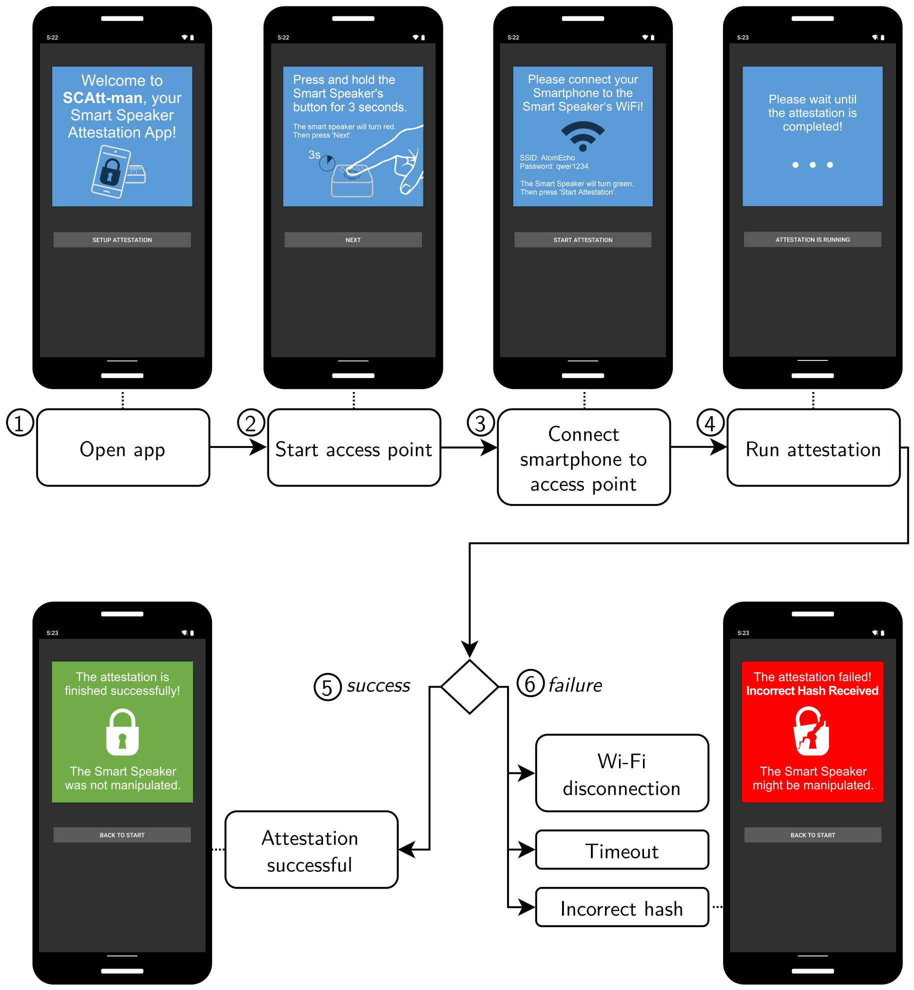

Welcome to SCAtt-man, your Smart Speaker Attestation App!
SETUP ATTESTATION
Press and hold the Smart Speaker's button for 3 seconds.
The smart speaker will turn red. Then press 'Next'.
3s
NEXT
Please connect your Smartphone to the Smart Speaker's WiFi!
SSID: AtomEcho
Password: qwer1234
The Smart Speaker will turn green. Then press 'Start Attestation'.
START ATTESTATION
Please wait until the attestation is completed!
ATTESTATION IS RUNNING
1 Open app
2 Start access point
3 Connect smartphone to access point
4 Run attestation
5 success
6 failure
The attestation is finished successfully!
The Smart Speaker was not manipulated.
BACK TO START
Attestation successful
Wi-Fi disconnection
Timeout
Incorrect hash
The attestation failed! Incorrect Hash Received
The Smart Speaker might be manipulated.
BACK TO START

Figure 4.7: The usage of SCAtt-man. The user is guided through the attestation process. In the end, the smartphone displays whether the attested device was verified successfully or if the attestation failed.

4.5.4 Design of the Attestation App

To assure that the app is usable for most users, the implementation process followed the guidelines from existing literature [48, 196, 217]. To use the screen space efficiently, the user is guided through the attestation process in several steps. Each step describes a single task in large font size with an additional picture or icon to aid the user's understanding and lower the threshold to use the app. To navigate to the next step, a button is provided at the bottom of each instruction labeled with the purpose of the next step. The color design supports the attestation process by changing the background color during the steps. The instructions are presented on a neutral light blue background while the results are shown on backgrounds with signal colors green for a successful attestation and red indicating a compromised device. For color-blind persons, the final result is accompanied by the icon of a lock signaling the integrity of the system or a broken lock indicating a tampered system or a failed run.

4.5.5 Usage Process of SCAtt-man Attestation

The usage process is depicted in Figure 4.7 with screenshots. Furthermore, we provide a video showing the full functionality[8]. The SCAtt-man attestation app guides the attestation process. To start the attestation, the user opens the attestation app.

1. The app shows a welcome screen explaining the functionality. The user presses 'Setup attestation' to start the attestation.

2. The app asks the user to press the button on the smart speaker for 3 s. The status LED on the smart speaker changes to red to indicate the start of the attestation process. Now, the attested device starts an access point.

3. The app asks the user to connect the smartphone with the newly started Wi-Fi access point of the smart speaker. As soon as the smartphone is connected, the status LED of the smart speaker switches to green. In the background, the attestation app now checks the continuous connectivity with the smart speaker.

4. With a click on 'Start attestation' the user starts the attestation process. The smartphone sends the nonce via sound to the attested device. The attested device receives this nonce and then runs the attestation function. As soon as the attestation function is completed, the result is sent via sound to the smartphone. The smartphone app receives this response and then compares this attestation response with the benign state. Furthermore, the app checks the time until the result is received.

[8] `https://youtu.be/HEbm7crMCU8`

5. If the response contains the correct measurements and arrived within the time threshold t_s, then the green screen with a closed lock is shown, signaling a successful attestation.

6. If the response took longer than a threshold value t_s, the app shows an error message on a red screen with a broken lock. This indicates a failed attestation. An error message is also shown if the result of the attestation function does not match the expected value or if the Wi-Fi connection to the smart speaker was interrupted.

If the attestation fails, the user can restart the attestation function. Attestation failure may be caused by transmission error, loss of Wi-Fi-connection, or actual compromise. The error message explains the potential reason for the failure. Furthermore, since the user perceives the communication, the user can detect distortions, such as loud noises, and restart the attestation process. In case of transmission errors or connection problems, users just need to repeat the attestation process. However, in our experiments, transmission errors and connection problems rarely occurred, as shown in Sections 4.6.2 and 4.6.4. Repeated failures indicate a real compromise. The app should then guide the user through a restoration process, for example, by resetting the device to resolve a compromise.

4.5.6 Integration Guidelines

The integration of SCAtt-man is straightforward. The most important task is to select a suitable side-channel for the communication and use this channel to implement a data transmission protocol. This side-channel can be sound, as used in our implementation, but for instance, also light. Smartphones typically feature a camera and a LED as a flash to brighten up photos which can be used to implement the communication with a smart light bulb that features a brightness sensor. Similar to Section 4.6.2, optimizing this new transmission protocol to achieve high reliability is necessary. The attestation function in SCAtt-man is similar to RealSWATT. Likewise, it is possible to configure its runtime to dominate the transmission time of the attestation request and response as demonstrated in Section 4.6.1.

We opt for the real-time operating system (RTOS) FreeRTOS for the implementation. This system allows management of the different tasks on the attested device, including all attestation and data transmission tasks. It is also possible to Integrate SCAtt-man on devices that do not feature an operating system, so-called bare metal systems. However, because one cannot rely on the abstraction and functionality given by a real-time operating system, the integration will need to be significantly more rigorous.

Table 4.1: Runtime of attestation function depending on the number of repetitions. All measurements are taken in ms.

Rep.	Min./Max.	Median	Average	Var.	SD
1	812.4/812.9	812.433	812.453	0.006	0.080
2	1624.8/1624.9	1624.927	1624.898	0.003	0.051
3	2437.2/2437.4	2437.372	2437.347	0.005	0.070
4	3249.6/3249.9	3249.769	3249.800	0.016	0.126
5	4062.0/4062.4	4062.152	4062.247	0.023	0.153
6	4874.4/4874.0	4874.559	4874.737	0.058	0.241
7	5686.9/5687.6	5686.980	5687.108	0.045	0.212
8	6499.2/6499.8	6499.398	6499.569	0.066	0.256
9	7311.7/7312.4	7311.824	7312.061	0.113	0.336
10	8124.0/8124.7	8124.249	8124.401	0.074	0.272

In the next section, we show the applicability of SCAtt-man in an end-to-end example and show how we optimized the parameters of the data transmission via sound.

4.6 Evaluation

In this section, we show that SCAtt-man is capable of performing a secure and reliable attestation. Furthermore, we explain how we determined the parameters for the data-over-sound protocol. In Section 4.6.4 we show how SCAtt-man can detect real-world attacks in a case study.

4.6.1 Runtime of Attestation Function

The runtime of the attestation function is the main security feature of SCAtt-man attestation, as elaborated in Section 4.4. Therefore, we performed a measurement study on our implementation of SCAtt-man on the smart speaker to obtain the runtime of the attestation function. These measurements are crucial in determining the timing thresholds for software-based attestation. The attestation function can be tuned to adapt its runtime to requirements by changing the number of iterations. Each additional iteration increases the runtime of the function. Due to the construction of the attestation function the implementation cannot be parallelized. More details on the design and the implementation of the attestation function can be found in Section 4.4.2. We conducted an extensive measurement study on the runtime of the attestation function and report the results in Table 4.1. We tested one to ten repetitions, yielding an average runtime of 0.81 to 8.1 s; repeating each test 158 times. Our measurement shows that the variance of the runtime of the attestation function is marginal, with a maximum at nine repetitions and a variance of 113 µs. This experiment shows that the attestation function is well-suited for performing software-based attestation.

4.6.2 Designing a Reliable Audio Protocol

As alluded to earlier, the communication protocol must allow reliable communication between the prover and the verifier. When the attestation fails, the user cannot determine the cause of this failure: The user cannot distinguish between a failure due to a compromised device and a failure due to a communication error. Thus, it is important to carefully develop the audio protocol such that it allows reliable attestation. The data-over-sound protocol has many parameters that have to be tuned to fit this use case. We conducted an extensive study to determine the optimal values for these parameters. In particular, these parameters are (1) the block length, i.e., the duration of each tone, (2) the base frequency, i.e., the frequency of each tone, and (3) the frequency separation, i.e., the difference in the frequency between simultaneously transmitted bits.

Based on our experiences from the implementation, we limited the state space to a block size of 100–600 ms (steps of 100 ms) and a frequency separation of 100–800 Hz (steps of 100 Hz). We observed that a slight offset of the base frequency by 10 Hz improves the transmission quality mainly because it reduces conflicting overtones. Therefore, we chose a base frequency of 510–2010 Hz (steps of 250 Hz). In this test, the smart speaker sends random messages to the smartphone, as this is the most tricky part due to the speaker of the ATOM Echo. As receiving device we used a Xiaomi Redmi Note 10. We found that transmissions work best with a block length of 240 ms. We extensively tested the transmission quality between the smartphone and the smart speaker.

Parameters for Audio Protocol

We performed an extensive parameter study to find the optimal audio protocol parameters. In the first round, we tested the base frequency and separation frequency. We chose a fixed block length of 500 ms to reduce the initial state space and limit the number of tests to be performed, assuming that a longer block length increases quality. The smart speaker sends ten random messages to the smartphone for each configuration. Figure 4.8 shows the number of correctly transmitted messages. We observed that a base frequency of less than 1010 Hz does not work.

In a second round, we evaluated the seven best configurations of the first round. In each of these cases, at least 70% of the messages were transmitted correctly. We increased the number of iterations of every configuration to 20. Thus, for each of the

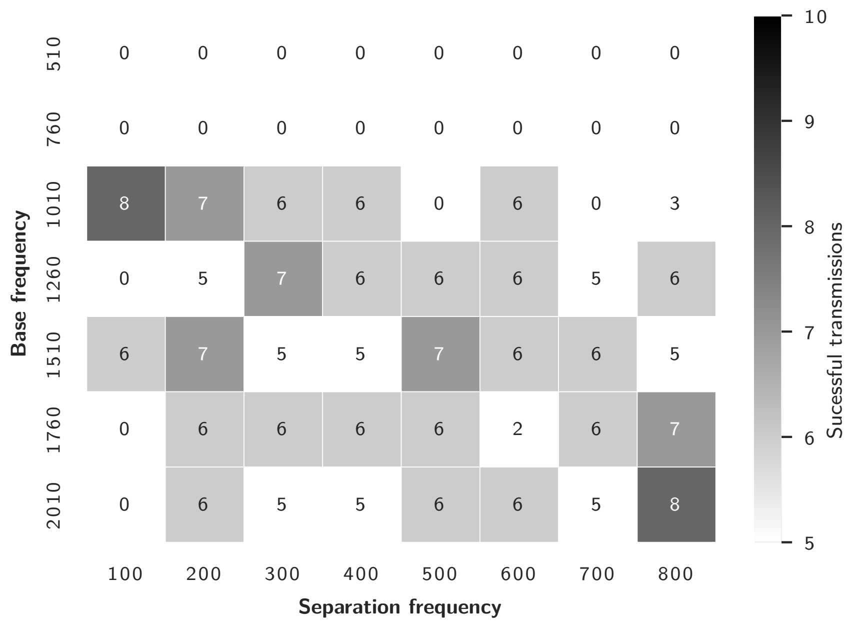


Figure 4.8: Evaluation of the base- and separation frequency.

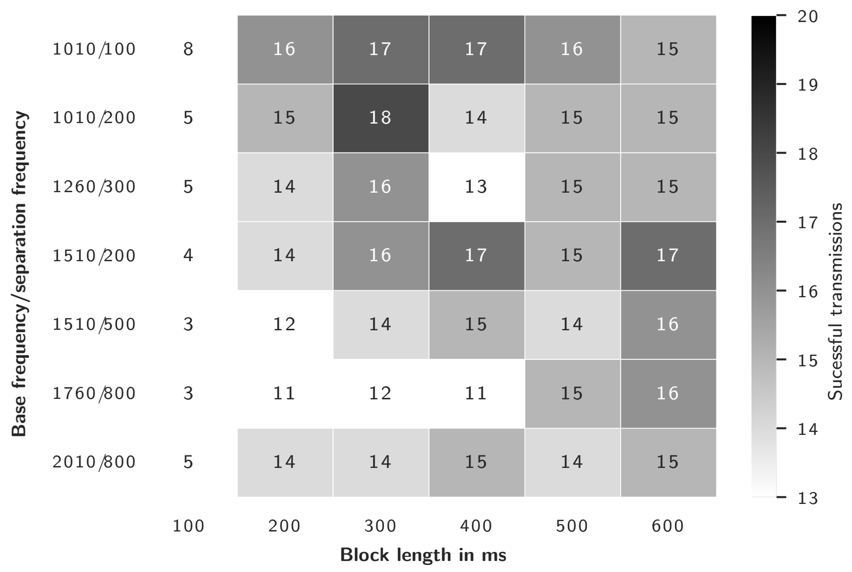


Figure 4.9: Evaluation of the base- and separation frequency and block length.

Table 4.2: Transmission success rate.

Direction	Success rate
ATOM Echo → Smartphone	96.5%
Smartphone → ATOM Echo	99.9%
Smartphone → Smartphone	100.0%

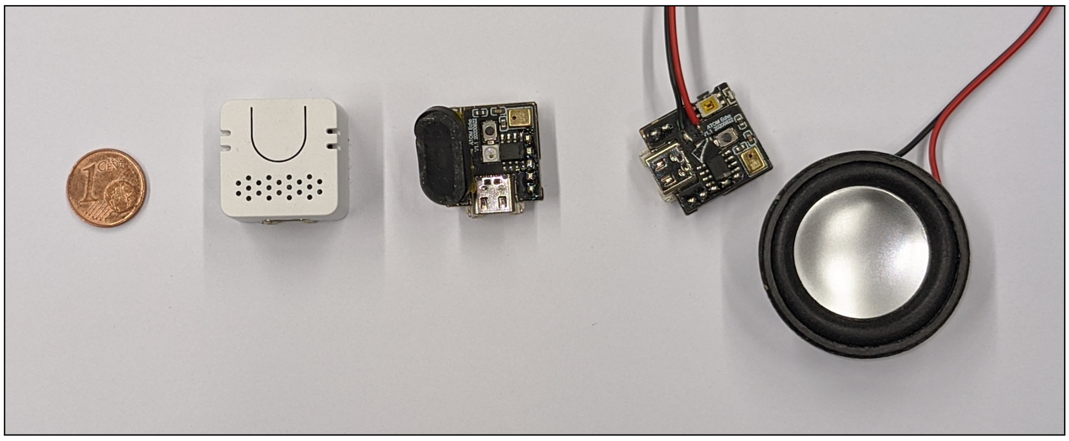


Figure 4.10: Modifications of the ATOM Echo. Left: The original ATOM Echo. Center: The ATOM Echo without casing. Right: ATOM Echo with the new speaker. Note the coin for scale.

seven configurations, we tested six block lengths with 20 transmissions each, a total of 840 messages. The results in Figure 4.9 show that short transmissions with 100 ms do not work well. We obtained the best results with a base frequency of 1010 Hz, a frequency separation of 200 Hz, and a block length of 300 ms. In further tests, we systematically analyzed the transmission of all possible bit patterns. We found that transmissions work best with a block length of 240 ms. In the following, we check the transmission quality between the smartphone and the smart speaker in extensive tests.

4.6.3 Further Audio Optimizations

During our experiments, we found the speaker of the M5Stack ATOM Echo to be a limiting factor. Its small form factor and low maximum volume frequently caused transmission faults. So we performed an experiment to verify this assumption, comparing the successful transmission in both directions between the ATOM Echo and the smartphone. In addition, we tested the transmission between two smartphones, a Xiaomi Redmi Note 10 and a Samsung Galaxy A6. In total, we sent

100 transmissions, consisting of eight blocks with 4 bit each, systematically testing all 2^4 bit possibilities. Table 4.2 shows the results: while all transmissions between the two smartphones were correct, and only one message received by the ATOM Echo was incorrectly transmitted. However, only 96.5% blocks sent by the ATOM Echo were correctly received. In total, 23% of all sent messages, each consisting of four blocks, were corrupted during transmission from the ATOM Echo to the smartphone. This confirms our assumption that the speaker of the ATOM Echo is the limiting factor. Hence, we first removed the microcontroller and the speaker from the plastic casing. As this did not sufficiently increase the transmission quality, we replaced the speaker with a larger model. Note that we did not change any hardware on the microcontroller. The amplifier, microphone, and processor stayed the same. Using the larger speaker, we yielded a success rate of 100% for the transmission between the smart speaker and the verifier. Figure 4.10 shows the changes to the ATOM Echo. Note the coin for comparison. The ATOM Echo has a small form factor of $24 \times 24 \times 17$ mm^3. In further evaluation, we used this larger speaker.

All of our tests were conducted in a typical office environment. We found that noise disturbances, such as people speaking or traffic noises through the window, did not influence our processes. We attribute this to the use of specific frequencies. We conclude that our attestation method works reliably in typical home and office environments.

4.6.4 End-to-End Case Study

We performed a case study to evaluate the full functionality of SCAtt-man. We used a Google Pixel 3 to run the verifier app and the ATOM Echo with the improved speaker. To show how SCAtt-man detects real-world compromises, we integrated a vulnerable configuration web interface into the smart speaker. The web interface allows to change the configuration of the smart speaker. This is a typical vulnerability, as the most common weaknesses in IoT devices are weak, guessable, or hard-coded passwords and insecure network interfaces and services [208]. The web interface allows one to change the URL for the speech-to-text service, that is, the service to which the user's voice commands are being sent. This poses a serious risk to the user's security and privacy. An attacker-controlled speech-to-text service allows recording a user's voice commands or arbitrarily altering the commands that the smart speaker executes. This URL is stored in the NVS partition, where all the smart speaker configuration data is kept. As soon as the user starts a speech command, the HTTP client reads this information, which performs the communication with the speech-to-text service. However, as soon as the URL is updated, this changes the content of the NVS partition. Hence, this is detected by the attestation. To check this, after altering the URL via the web interface, we start an attestation run

Table 4.3: Results of end-to-end case study.

		Attestation result	
		Benign	Compromised
System state	Benign	50	0
	Compromised	0	10

which correctly detects the modifications. We repeated the attestation run ten times. Every time the attestation failed correctly, that is, the compromise was detected.

This full end-to-end example shows how SCAtt-man successfully checks the integrity of the smart speaker's configuration and is able to detect compromises like malware or altered configurations. To show the reliability of this attestation, we manually repeated the attestation multiple times with the benign and the compromised device. In total, we performed the attestation 50 times on the benign device and ten times on the compromised device. Table 4.3 shows the result. All benign and compromised states were successfully identified. There were no false positives or negatives.

4.6.5 User Study

To ensure a sufficient usability of SCAtt-man we conducted a qualitative user study consisting of two parts. First, the users were asked to interact with the SCAtt-man smart speaker and perform the attestation process using a Pixel 3 smartphone. Second, the users completed a set of questionnaires. The full results of the questionnaire are shown in Table 4.4. We recruited 20 participants among company personnel and university students for the study, of which 6 identified as female and 14 as male. The age of the participants ranged from 20 to 65 years with a mean of 37.7. All of them participated voluntarily and no compensation was paid. Each participant was informed about the study objective before the study and signed an informed consent explaining which data was collected and how it would be processed.

Literature indicates that 20 participants are sufficient to identify at least 95% (mean 98.4%) of all usability problems [108]. Our results did not show a large deviation between users' responses, indicating that saturation was reached. The lack of new input from users signals a sufficient sample size for qualitative studies [52].

Since all participants were recruited among company staff and on the university campus, participants were also tasked with filling out the ATI-Scale (Affinity for Technology Interaction) [113] questionnaire, in order to make this sample comparable to other studies. The results show that a mean of 4.43 was reached with a standard error of the mean of 1.01 and a Cronbach's alpha of 0.8. To assess the users' previous knowledge regarding smart home technology, a set of seven questions was

Table 4.4: Full results of the questionnaire.

	Age (years)	27	26	25	34	32	27	20	27	31	30	36	51	37	48	46	47	49	53	43	65
	Gender (f/m)	m	m	m	f	f	m	m	m	m	m	m	m	f	m	m	f	m	m	f	f
Questionnaire	I used smart speakers before.	3	1	6	6	2	2	4	1	4	1	1	2	1	1	6	1	1	6	6	1
	I trust IoT devices like smart speakers.	5	2	3	6	1	1	3	2	1	3	3	3	5	4	3	2	1	4	3	2
	I trust my smartphone.	5	3	3	4	3	5	2	3	2	4	4	4	4	6	4	3	4	5	5	6
	Additional security measures can increase my trust in IoT devices.	6	6	5	6	4	5	5	6	4	5	6	6	6	6	6	6	6	6	6	2
	I believe that attestation can detect manipulations in devices.	4	6	5	6	0	5	5	6	4	5	5	4	4	5	5	6	6	5	6	4
	The audio communication increases my trust in attestation.	4	4	4	5	6	5	3	5	4	5	4	4	2	6	5	3	6	5	5	3
	When an IoT devices has an integrated attestation method I would use this functionality.	5	5	5	6	0	6	6	6	6	6	5	3	4	6	6	6	6	6	4	6
ATI	I like to occupy myself in greater detail with technical systems.	5	5	6	4	4	6	4	4	6	5	6	4	3	6	5	5	5	5	4	4
	I like testing the functions of new technical systems.	4	6	6	5	5	6	4	5	6	6	6	6	3	6	5	5	6	6	3	2
	I predominantly deal with technical systems because I have to.	4	4	1	4	3	1	4	5	1	3	3	5	4	6	3	3	5	6	4	5
	When I have a new technical system in front of me, I try it out intensively.	4	5	6	5	5	6	5	4	5	5	6	6	4	6	5	4	6	6	3	2
	I enjoy spending time becoming acquainted with a new technical system.	3	5	6	4	3	6	4	4	6	5	6	5	3	6	5	5	6	6	3	2
	It is enough for me that a technical system works; I don't care how or why.	2	2	2	3	1	1	3	3	2	1	2	3	4	2	1	4	2	1	6	6
	I try to understand how a technical system exactly works.	4	2	5	4	5	6	4	2	5	6	6	4	3	5	5	4	6	5	2	1
	It is enough for me to know the basic functions of a technical system.	5	3	2	4	1	1	3	3	1	2	5	4	5	5	2	4	2	1	4	6
	I try to make full use of the capabilities of a technical system.	4	4	6	5	6	6	5	4	5	5	5	2	5	4	5	5	5	6	3	3
UEQ-S	obstructive — supportive	5	6	7	7	7	7	5	6	5	6	7	5	4	5	6	5	6	6	5	7
	complicated — easy	6	7	7	7	7	6	7	7	5	6	4	4	5	3	7	5	7	7	3	7
	inefficient — efficient	5	5	7	7	7	6	6	7	4	5	6	6	5	5	7	5	6	7	4	7
	confusing —- clear	6	7	7	7	7	5	7	7	6	6	6	6	3	4	6	5	5	7	5	7
	boring —- exciting	7	4	5	7	7	6	5	5	4	4	6	7	4	4	4	5	5	6	4	6
	not interesting — interesting	6	5	5	7	7	7	7	7	5	4	7	5	5	6	6	5	6	7	5	6
	conventional —- inventive	4	5	6	7	7	6	5	5	5	7	5	5	5	6	7	5	7	6	5	6
	usual — leading edge	5	4	7	7	7	7	5	5	4	6	5	7	4	5	5	6	7	6	4	6

Table 4.5: Responses to the questionnaire on SCAtt-man. Answers range from (1) Completely disagree to (6) Completely agree.

	Question	Average	Median	SD
Q1	I used smart speakers before.	2.80	2	2.11
Q2	I trust IoT devices like smart speakers.	2.85	3	1.42
Q3	I trust my smartphone.	3.95	4	1.15
Q4	Additional security measures can increase my trust in IoT devices.	5.40	6	1.05
Q5	I believe that attestation can detect manipulations in devices.	4.80	5	1.36
Q6	The audio communication increases my trust in attestation.	4.40	4.5	1.10
Q7	When an IoT device has an integrated attestation method I would use this functionality.	5.15	6	1.50

included in the questionnaire. Table 4.5 shows those questions and the answers. In addition, participants were asked to indicate the degree to which they agree/disagree with the statements on a 6-point Likert-scale and a "not applicable"-option, so choosing from the following options: (1) Completely disagree, (2) Largely disagree, (3) Slightly disagree, (4) Slightly agree, (5) Largely agree, (6) Completely agree, (7) Not applicable.

The results showed that more than half of the participants had little or no experience with smart speakers (mean 2.8) and that they trusted these devices less than their smartphones (2.85 compared to 3.95). The four items about trust in attestation and its use were significantly positive (5.4; 4.8; 4.4 and 5.15). In particular, the value of 5.4 shows that trust in smart speakers can be enhanced using additional security techniques. The participants stated that attestation techniques detect manipulations in devices with a rating of 4.8. In particular, the users think that the observable audio communication further increases the trust in the attestation scheme (4.4). When an IoT device would have an attestation function, the participants would use it (5.15).

During the interaction with SCAtt-man the users should speak out their thoughts according to the 'think-aloud' method to identify possible problems in the usage process. Our study demonstrates that the participants considered our app highly usable. However, users tend to always click the button to proceed to the next step, ignoring the task (for example, pressing the button on the smart speaker, connecting the Wi-Fi), resulting in a failed attestation, even though they read the instructions. Often, they realized their mistake afterwards and used the back-function of the attestation app to repeat the previous step. Furthermore, we observed that manually changing the Wi-Fi configuration is tricky due to the many vendor-specific implementations. To further increase the usability of the SCAtt-man app, we plan to integrate checks preventing continuation before completing the respective task.

In order to assess the usability of the system, the UEQ-S questionnaire (User

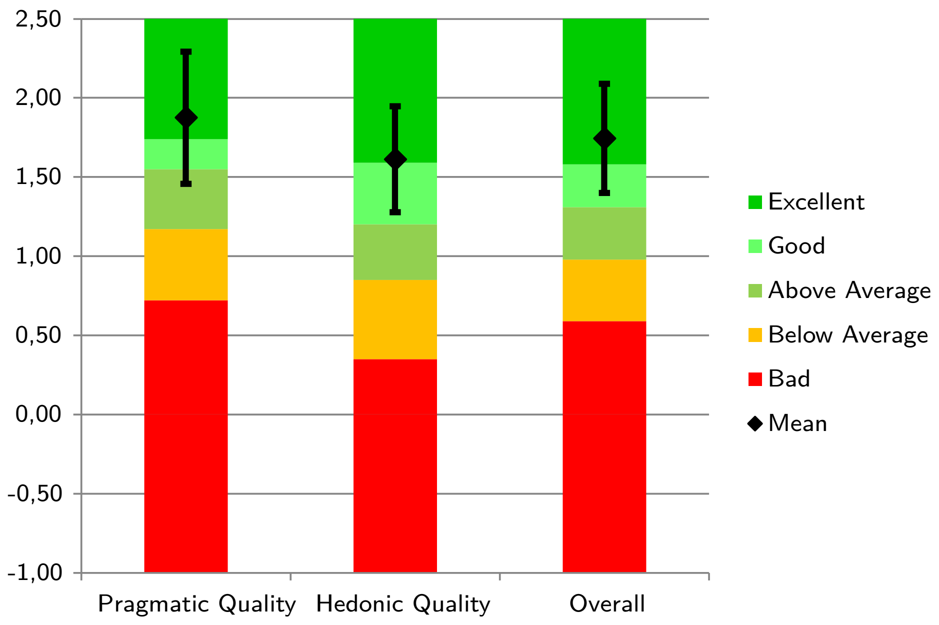


Figure 4.11: Results of the UEQ-S questionnaire.

Experience Questionnaire, short version) [235] was filled out after the hands-on experiment. Figure 4.11 shows our results. The results show that the usability of the system was rated positively: the overall score was 1.719, the pragmatic quality was rated 1.875, and the hedonic quality with 1.563. Since all three values lie above the threshold of 0.8, the usability of SCATT-MAN was evaluated positively.

4.6.6 Summary

We conclude that SCATT-MAN fulfills the most significant requirements towards a usable attestation scheme. SCATT-MAN has both good usability and users trust the attestation. Furthermore, if devices featured an attestation method, users would use it. In an end-to-end example, we showed how SCATT-MAN is able to reliably detect a real-world attack on a smart speaker.

4.7 Security Discussion

A secure software-based attestation scheme requires careful design and implementation. In the following, we discuss typical attacks on software-based attestation and explain how SCATT-MAN addresses these.

Time-of-Check/Time-of-Use. Existing attestation schemes are vulnerable to time-of-check/time-of-use (TOCTOU) attacks [254]. The key idea of this attack is to restore a benign state before the next attestation run, allowing the attacker to stay undetected. However, in SCAtt-man, the attacker cannot predict the attestation. The user randomly triggers attestation runs through physical interaction (for example, a button press). Since the attacker cannot predict the user's attestation request, the attacker cannot restore the benign state in time. Thus, the malicious behavior will be detected over time.

Network Delays. For software-based attestation, it is crucial that the transmission time can be predicted precisely [241]. As such, software-based attestation is usually limited to one-hop settings [32]. When using shared communication channels such as Wi-Fi or other radio transmissions, the attestation needs to deal with network delays, for example, due to interference with concurring transmissions, devices, or other disruptions. Short-distance communication over light or sound has a predictable transmission time due to the direct communication from device to device. The user is able to observe the transmission and identify disturbances, for example, background noise that disturbed the transmission.

Offloading Attacks. As communication happens directly over a side-channel such as light or sound, there is no need for a remote network connection. Therefore, the smart speaker can implement a network disconnection, for example through a physical hardware kill switch or other mechanisms, which efficiently prevent any network connection during attestation as described in Section 4.4.3. Thus, there is no network connection to a third party to which the attested device can relay its communication. Consequently, attestation reports cannot be forged.

Compression Attacks. A compression attack uses a compression mechanism to free up memory on the attested device, which is not covered by the attestation and may store malicious code [241]. The attacker can perform this attack on-the-fly, compressing and decompressing memory segments on demand to avoid detection. However, this strategy only works if three preconditions are met: (1) The attacker knows that attestation has started. (2) The attacker can predict which memory region is currently being attested in order to copy from one region to another. (3) The attacker has sufficient resources to move the data during attestation.

SCAtt-man addresses such compression attacks as follows. (1) The user randomly chooses to perform the attestation. However, cutting off network communication may be regarded as a signal of the start of the attestation. (2) In order to make compression attacks more difficult we adopted several techniques from previous work, see Chapter 3. First, we start the attestation process at a random location, derived from the nonce. Second, we fill the empty memory with random data to make compression itself difficult. (3) Writing data on embedded devices has a high latency since write operations need to rewrite entire pages (sections) of memory. Those hardware limitations are described in Chapter 3. Furthermore, the ESP32 and most IoT devices offer hardware acceleration for hashing. Standard hashing algorithms such as SHA-256 use a Merkle–Damgård [182] construction [214]. The output of the previous block is used as the input of the next block. Due to the high speed of hashing (using hardware acceleration) and the hardware restrictions on write operations (rewriting flash pages is slow), compression attacks become infeasible for the attacker.

Memory Manipulation. Compression attacks are more sophisticated than simple memory manipulations. In a memory manipulation attack, the attacker uses unmonitored memory such as the RAM to avoid detection. However, due to already existing countermeasures and embedded hardware characteristics, this kind of attack has often become infeasible on embedded devices. For example, RAM is typically marked as non-executable memory [101]. Therefore, no code can be executed from this memory region. Furthermore, manipulating the memory layout, such as the partition table is highly challenging [102]. The ESP32, which we use for our proof-of-concept smart speaker implementation, enforces several preconditions to change the existing memory layout. First, SPI Dangerous Write must be explicitly enabled. Second, such changes are typically implemented by OTA (Over-the-air) updates [100]. This would require a full reboot, and in addition it would change the stored partition information [102]. Thus, the user-invoked attestation will fail.

Attacks on the Attestation Mechanism. Since SCAtt-man uses a side-channel for attestation, the network communication is completely turned off during attestation as already described in Section 4.3.2. Thus, leaving only direct attacks on the attestation protocol. Since the attacker can only start attestation once the nonce is received, the critical point is the time span between the reception of the last message containing the nonce and the beginning of the transmission of the response to the verifier. A malicious actor will likely try to speed up the recognition of the last message to gain some time for malicious code execution, as the attestation function by itself cannot be accelerated: The attestation function consists of a non-parallelizable hashing function with an optimized or even hardware-accelerated implementation. More details on the implementation of the hashing function can be found in Section 4.5.2. The verifier transmits each message for 240 ms, thus the attacker may try to recognize the last message earlier. In Section 4.5.3 we already described that our ESP32 implementation listens for 32 ms due to resource constraints. Furthermore, we evaluated the time needed for fast Fourier Transformation (FFT) to analyze the recorded sound fragments. In order to avoid speedup of the nonce recognition, we have considered the minimal recording and processing time to receive the complete nonce. Based on those recording and processing times we carefully set the threshold for attestation. Consequently, there is no usable time gap in the data-over-sound transmission. The threshold needs to be adjusted on a per-device basis, based on the available resources and sound processing capabilities of the device.

4.8 Related Work: Context-Based Authentication

Context-based authentication is complementary to SCATT-MAN attestation.

Secure key exchange and device authentication is a hard problem, especially for large settings with many devices, and embedded devices that do not have a user interface. The goal of context-based authentication is to automatize the paring process of devices using measurements of the environment. It bases on the general assumption that devices that belong to each other are close together and share the same context.

Context-based authentication allows different parties within the same context, such as devices within the same room, to identify and authenticate each other and negotiate a shared key, for example, to establish a secure connection. Similar to SCATT-MAN attestation, context-based authentication uses short-range communication channels to pair devices or negotiate keys with nearby devices. While SCATT-MAN uses a data-over-sound protocol there are numerous systems for context-based authentication that work with ambient noise. Context-based authentication schemes describe which context features are measured and how shared keys are derived from these features. Research proposed a wide variety of such schemes.

PINtext is a context-based authentication scheme based on ambient sound. Devices send each other a pairing request and negotiate a start time for the audio recording. The devices then derive a fingerprint from this audio sequence [236]. GAB-IoT is a group-based authentication scheme based on sound. It uses ambient sound to exchange keys, but users can provide additional sound to enhance the process. The focus of GAB-IoT is on usability and user interaction. Users can control the pairing process and confirm devices using a smartphone app [129]. Another approach is Listen!. It uses a similar setup as GAB-IoT, but adds additional noise to speed up the pairing process [179].

Gait-Key has a different setting. It uses gait patterns of users to pair personal devices, such as body sensors [289]. Instead of using sound, Gait-Key uses a person's walking style to authenticate devices. This makes Gait-Key suitable to pair devices on the body of a user, for instance, biomedical sensors in a body-area network [135].

There is a major limitation of all of these context-based authentication schemes: All devices require the same sensor type to take measurements. Therefore, pairing between devices with different sensor types is not possible. In contrast, Perceptio allows pairing between devices with different sensor types, for example microphones, accelerometers, and a motion detectors. This is achieved by basing pairing on events that can be perceived between different types of sensors. Similar to traditional context-based authentication schemes, services outside of the shared context cannot consistently monitor these events. Inside the same context, events can be monitored and related to each other. Events are typically caused by the environment, such as household members. To reduce the pairing time also a signal injection device may be introduced [134].

These context-based authentication techniques have also been adapted to other use cases. For instance, Sound-Proof allows two-factor authentication with smartphones based on ambient sound. It compares ambient sound between devices where users are logging in, such as a notebook, and a second device, such as a smartphone. Compared to other two-factor-authentication approaches, Sound-Proof does not require any user interaction with the phone [153].

4.9 Summary and Conclusions

In this paper we presented SCAtt-man, a solution to perform secure software-based attestation on IoT devices. SCAtt-man solves the inherent problem of missing device authentication in software-based attestation by using user-observable side-channels. This approach allows the user to identify the attested device. We implemented SCAtt-man into a smart speaker and developed an app for Android smartphones to perform the attestation. The Android app guides the user through the attestation process. Our evaluation shows that SCAtt-man can reliably perform attestation without failures. In a full end-to-end example, we showed how SCAtt-man can be used to detect a compromise caused through a typical real-world vulnerability. In a user study, we found not only that SCAtt-man has a good usability, but also that people trust attestation solutions in general and would use them if their devices had such a feature. This makes novel attestation solutions for customer IoT devices a worthwhile research target.

The concept of side-channel-based authentication can also be applied to other side-channels beyond sound. For example, light could be used on IoT devices that feature a lamp and a brightness sensor, in combination with the flash and camera of a smartphone. For future work, we will perform further extensive user studies to investigate the usability and user experience as well as understanding of SCAtt-man attestation. In particular, the trust in tools like SCAtt-man and the understanding of the underlying process are crucial for deployment into practice and have to be investigated beforehand. Therefore, integrating end users into the design process of attestation tools is a promising approach to avoid reluctance by potential customers.

CHAPTER 5

DMA-Based Remote Attestation

As shown, remote attestation enables validating the integrity of a remote device, thereby establishing trust in the device. But existing attestation techniques either rely on trusted computer components, rigid temporal constraints, or hardware modifications. Hardware-based approaches require trusted computing modules such as ARM TrustZone to perform secure measurements of the attested device [3, 4], which are often not available on small and embedded devices. Hybrid approaches need custom hardware extensions that are expensive for initial implementation and unavailable on legacy devices [49, 201]. However, also software-only attestation schemes are only a partial solution. Software-based approaches do not have hardware requirements but rather rely on precise measurements of execution time and therefore have strict requirements towards their implementation and communication, limiting their practical applicability. However, the implementation is complex and error-prone [57]. Determining the correct timing threshold is a delicate task, and implementation flaws can undermine all security assumptions of the attestation scheme. Implementing such attestation schemes requires deep knowledge of the attested system and a specific implementation for each device and its corresponding configuration. This limits the applicability of software-based attestation on legacy devices, which have varying hardware and software configurations. In addition, the complex implementation typically also demands access to the device's source code, which is often unavailable for such devices. These aspects hinder the practical application of software-based remote attestation.

Furthermore, software-based solutions have conceptual limitations. For instance, they have the inherent problem of a lacking root of trust that allows a reliable device identification. There are different solutions to solve this. RealSWATT (see Chapter 3), for example, requires a dedicated processor core as well as a separate network and IoT gateway. In Chapter 4, we presented SCAtt-man, which uses side-channels to make the attestation process user-observable and enable the user to identify the attested device.

Summing up, while remote attestation is a viable solution to enhance legacy systems, integrating such techniques is often challenging because of missing hardware features, unavailable source code, and real-time constraints. During the development of real-time applications, their timing behavior has been extensively tested in a profiling phase. Any changes to these applications, even to introduce instrumentation for control flow integrity [1], influences the timing behavior, hence requiring this profiling phase to be repeated [79, 277].

Another challenge is the verification: A remote instance must decide whether the attested device is in a benign state or whether it has been compromised. Typical attestation protocols use hash methods to perform a measurement of the attested device. Verifying these hashes requires prior knowledge of all states, as a hash does not allow reconstruction of the input data. Depending on the complexity of the attested device, obtaining all benign states is an exhaustive task [2]. Furthermore, this exploration of valid states must be repeated upon changes to the device, making this approach inefficient and hindering practical application and updates of the attested device.

In remote attestation, the verifier typically continuously monitors the attested device. This leads to performance issues [3, 4, 201] and race conditions like the well-known time-of-check/time-of-use problem [83, 139]. In practice, verification of devices is desired, but continuous checking is not required. For example, Stuxnet was undetected for a long time and thereby able to cause severe damage [169]. Botnets like the Mirai botnet [7] inherently rely on undetected infections to maintain a large network of bots to fulfill tasks [25, 47]. When remaining undetected, these botnets can perform severe attacks. For instance, the Mirai botnet, which infected more than a million IoT devices, was responsible for one of the largest distributed denial-of-service (DDoS) attacks ever.

The relaxation of the verification time has many advantages in practice. Verifying the attestation reports of attested devices induces a great demand for communication to send these reports to the verifier. It poses a computational load on the verifier to run the actual verification process. Drastically reducing the number of verification runs allows for extending remote attestation to areas where this was not possible before. For example, in the case of a car or an autonomous drone, a continuous attestation of a large fleet of vehicles is impractical due to computing and communication limitations. Autonomous drones and cars are designed to operate independently without a permanent connection to another instance. Introducing a centralized attestation scheme removes this property. In addition, complex integrity checks require significant computing resources on the verifier side. However, more importantly, it is an impractical assumption that each vehicle always has uninterrupted communication. If the communication is disturbed or unavailable, attestation cannot take place. So, attestation requires reliable communication. Reliable communication is typically available in fixed locations. So, good moments for integrity checks are during existing service times or in the garage or hangar in case of vehicles, drones, or planes. In the case of industrial devices, attestation can be done by service technicians during regular maintenance. This way, the integrity of devices can be verified, and if a compromise is detected, it can be reacted immediately.

Besides the software itself, IoT devices also feature critical configuration data. Alterations of such configurations can have a severe impact. For example, in drones, often no-fly zones are configured [145]. In so-called data-only attacks, minimal changes of data in memory can have severe consequences, even without altering the program's control flow [64]. Therefore, it is essential to include this configuration data in the attestation.

All of these aspects limit the applicability of remote attestation on legacy embedded devices. A practical, universal solution for remote attestation to use on legacy devices would need to work without many changes to the device's firmware, ideally without requiring source code knowledge. The verifier should be straightforward to implement. Furthermore, the attestation scheme should offer a method for device identification.

Contributions. In this chapter, we present a novel remote attestation approach, called DMA'n'Play, that tackles the described practical limitations by leveraging DMA (direct memory access). Since DMA does not require CPU time, DMA'n'Play even allows attestation of devices with real-time constraints. To prevent the exploitation of side-channels which potentially could determine if the attestation is running, we developed another option called DMA'n'Play To-Go. This is a small, mobile attestation device that can be plugged into the attested device. We evaluated DMA'n'Play on two real-world devices, namely, a syringe pump and a drone. Our evaluation shows that DMA'n'Play adds negligible performance overhead and detects data-only attacks by validating critical data in memory.

DMA'n'Play allows verifying the integrity of devices during operation without requiring any trusted computing modules, hardware modification, or changes to the software of embedded applications. The general idea of DMA'n'Play is to enable the verifier to monitor the memory of the attested device using DMA directly. In traditional attestation schemes, trusted computing components or custom hardware extensions are used to perform a secure self-measurement. However, in the case of legacy embedded systems that do not feature such components, integrating these attestation schemes implies replacing the components of the embedded system. This is a costly and impractical process, as eluded earlier in Chapter 2. Here, DMA'n'Play is a viable solution: Instead of replacing the hardware components, we add a dedicated tiny, low-cost device to perform the attestation.

In addition, this also has a second advantage: DMA allows direct access to a system's memory without the involvement of the processor. DMA is typically used to speed up memory access of external devices and reduce the utilization of the processor. As the DMA controller is independent of the attested device, DMA enables trustworthy self-reports even on compromised systems. It is a standard feature of microcontrollers and is widely available in embedded devices as used in industry; supported by all major vendors including STMicroelectronics [258], NXP [206], and Infineon [147].

In contrast to traditional pure software-based attestation solutions, including SWATT [241], RealSWATT (see Chapter 3), and SCAtt-man (see Chapter 4), integrating the DMA'n'Play framework into existing applications is straightforward, as no complex runtime requirements have to be considered and no extensive execution time thresholds have to be provided. Moreover, DMA'n'Play is also suitable for timing-critical devices, for example, real-time or medical devices, where any change in hardware or software implies re-validation of the timing behavior. The implementation of DMA'n'Play does not change the software on the attested device. This makes the DMA'n'Play attestation framework also suitable for legacy devices as neither hardware modifications nor source code of the attested devices are required.

In DMA'n'Play, we use DMA to give an external device direct access to the main memory. This way, the external device can examine the main memory of another device during runtime. Using this direct access to main memory, the actual data in memory can be monitored, for example, variables and data structures. This enables DMA'n'Play to detect malicious manipulations on data in memory, detecting data-only attacks that traditional security techniques like control flow integrity cannot cover. We present a format for configuration that allows specifying the data structures to monitor and define constraints for valid states. The attested device cannot influence this investigation, as the DMA controller completely handles the memory access.

In contrast to traditional remote attestation schemes, DMA'n'Play requires the verifier to be directly connected to the attested device. In practice, depending on the setting, the verifier can either be a standard computer system, for example, a personal computer, smartphone, tablet, or an embedded device like a diagnostics terminal in a repair workshop. Additionally, we propose a tiny embedded device, dubbed DMA'n'Play To-Go, that can be used to relay the attestation measurements to a remote verifier or that can also be used directly as a verifier. For instance, DMA'n'Play To-Go can forward attestation measurements to an external verifier, for example, via a wireless transmission. This way, mobile devices like drones or vehicles can also be attested during operation.

In summary, we provide the following contributions:

- We propose DMA'n'Play, a novel remote attestation framework that uses DMA to monitor the attested device and specifically the content of its memory, thereby detecting manipulations and illegal states without requiring changes to either hardware or software, for example, reprogramming or instrumentation of the attested device.

- We show how the DMA'n'Play attestation framework uses a binary file of the attested device and a configuration file to define benign states of the attested device, allowing integration into both new and existing legacy devices.
- With DMA'n'Play To-Go, we present an external verification device that can be attached to attested devices to check their integrity continuously.
- We provide integration guidelines that show the necessary steps to implement DMA'n'Play attestation into devices.
- Following these guidelines, we integrated DMA'n'Play into two real-world systems, a medical device, and a drone, to show the general applicability.
- In a case study, we use DMA'n'Play to detect attacks on these devices: a manipulation of the injection rate of a syringe pump and modifications to the drone control system.

5.1 Background: U(S)ART, SPI, and DMA

In this section, we explain direct memory access (DMA) and give background on the standard serial communications protocols U(S)ART and SPI, that are used for DMA'n'Play.

5.1.1 U(S)ART and SPI

Microcontrollers usually interact with external peripherals such as sensors, displays, and control units. Hence, they need standardized interfaces such as UART (Universal asynchronous receiver-transmitter) [172] and SPI (Serial peripheral interface) [82] to exchange data.

The UART interface has two communication lines: one for transmitting data (TX) and one for receiving data (RX). UART is asynchronous, that is, it operates on a fixed clock cycle, which the receiver needs to be aware of to interpret the data correctly. USART (Universal synchronous and asynchronous receiver-transmitter) offers an additional clock signal, which is used to synchronize the transmitter and receiver. UART and USART communication is designed for direct bilateral device communication [82].

In contrast, SPI is designed for the communication of one single device (master) to multiple peripherals (slaves) like displays or sensors. SPI allows full duplex and

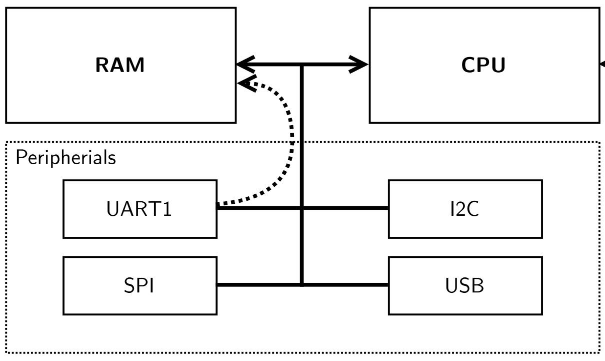


Figure 5.1: Architecture without DMA support.

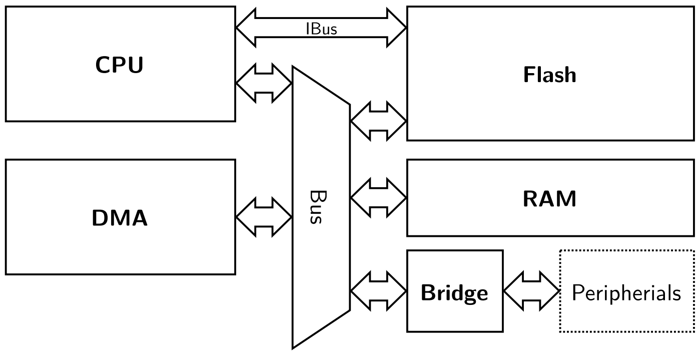


Figure 5.2: Architecture with DMA support.

synchronous communication using four lines: a serial clock provided by the master, data output from the master (MOSI), data output from the slave (MISO), and slave select ($\overline{\text{SS}}$) to indicate the master and slave configuration [187].

5.1.2 Direct Memory Access (DMA)

Direct memory access (DMA) is a common feature of microcontrollers that allows directly copying of memory contents from or to external peripherals. With DMA, the CPU does not need to manage data copying between RAM and peripherals. The purpose of DMA is to unburden the processor of the time-consuming task of moving data over the memory bus between places. The CPU does not need to poll for incoming or outgoing data transfers actively, and the CPU is not frequently interrupted (for example, by CPU interrupts) to move small bits of data [207].

DMA is a common and widespread feature on standard microcontrollers as used in the industry, supported by all major vendors, including STMicroelectronics [258], NXP [206], and Infineon [147]. Typically, the CPU is responsible for listening for incoming data from external peripherals (either through polling or interrupts) and copies the data into RAM. However, the DMA controller can be configured to directly copy data from or to an external peripheral like UART. The CPU now only needs to be notified, for example, by an interrupt once the copy routine has terminated. Note that a memory controller typically features multiple parallel DMA streams and offers precise predictability of execution times for real-time applications.

5.2 Challenges

As introduced at the beginning of this chapter, assuring the integrity of a remote system is a challenging task. The attested system itself is untrusted, so obtaining trustworthy reports from such a system is a complex problem. In Chapter 2.5 we explain the general challenges for secure remote attestation. From these, we derive the challenges that we need to tackle for secure attestation using DMA.
In particular, for the DMA'n'Play attestation scheme we need to tackle the following specific challenges:

Challenge 1: Secure Self-Measurement. As in any attestation scheme, we must guarantee that the attested device is not able to manipulate or delay the self-measurement.

Challenge 2: Detectability. When using an attestation approach that is not constantly running it is important that the attacker cannot detect whether the device is currently being attested. Otherwise, the attacker could hide or stop any attacks while the attestation is being executed. This includes both direct observations as well as side-channels, for example, monitoring other events that indicate an attestation.

Challenge 3: Root of Trust. In software-based remote attestation, any secret key on the attested device can be obtained upon full system compromise. The lack of a root of trust results in the problem that the verifier cannot distinguish between the genuine device, other devices, or simulations.

Challenge 4: Side-Effects and Real-Time Operation. Attestation may not negatively influence the normal operation of the attested system, especially when performing tasks with real-time constraints. This is especially important when integrating attestation into existing legacy devices, whose timing behavior have already been validated and hence may not be altered.

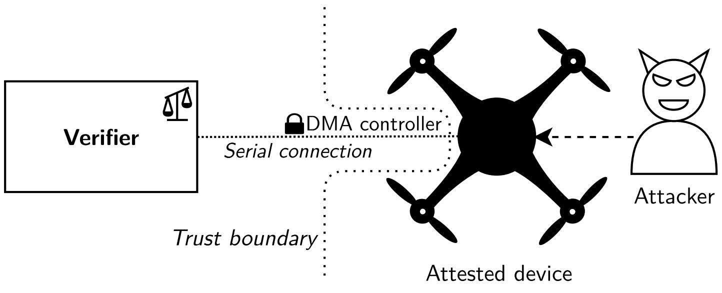


Figure 5.3: Threat model of DMA'n'Play with the verifier directly connected to the attested device. The adversary can fully compromise the attested device but cannot modify the verifier or the DMA controller.

Challenge 5: Efficient Verification. Implementing an efficient and effective verifier is a challenging task. Typical attestation protocols send hashes of the system's current state, from which the actual state cannot be reconstructed directly. The verifier requires a full lookup table of all benign states. Generating such a table is a complex task and requires a large amount of memory.

In the following, we will show how DMA'n'Play allows remote attestation while addressing these challenges by utilizing a standard DMA interface.

5.3 Assumptions and Threat Model

Figure 5.3 shows our threat model for DMA'n'Play as well as the trust assumptions. The attacker can fully compromise the attested device. However, the attacker cannot compromise the external verifier or the DMA'n'Play To-Go and cannot alter the configuration of the DMA controller.

5.3.1 Assumptions

We assume an embedded device that features a DMA controller that allows copying memory content to an external bus, such as a serial bus like UART or SPI. The ability to copy memory content to an external bus is a common feature of DMA controllers deployed on various embedded devices [147, 206, 258]. Furthermore, we assume that the configuration of the DMA controller can be locked, for example, by utilizing a memory protection unit (MPU), preventing the DMA configuration from being changed even when the entire system is compromised. This is also a widespread feature on existing microcontrollers [142, 146, 261]. We describe in detail how this can be achieved in Section 5.4.4.

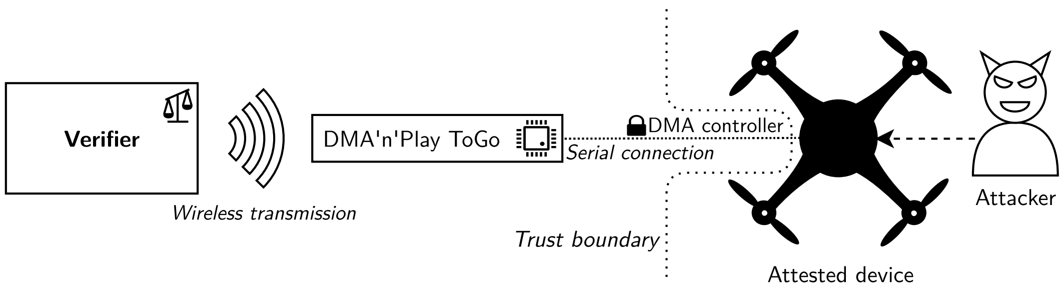


Figure 5.4: Threat model of DMA'n'Play with DMA'n'Play To-Go. The adversary can fully compromise the attested device but cannot attack the verifier, DMA'n'Play To-Go, or the DMA controller.

Second, as in any remote attestation scheme, we require a trusted verifier. During attestation, the verifier must be attached to the serial bus. This case is sketched in Figure 5.3. The verifier device does not have to be connected all the time but is only required during attestation time. The role of the verifier can be taken over by a commodity computer system, such as a notebook or workstation. The verifier can also be integrated into a diagnostics system typically used for repair and maintenance in the automobile space. Alternatively, we developed DMA'n'Play To-Go, a dedicated, small, low-cost embedded device to take over the role of the verifier and either perform the verification directly or relay the information to a remote verifier. Figure 5.4 shows this scenario: DMA'n'Play To-Go is directly attached to the attested device and transmits the measurements to the external verifier via a wireless communication channel. This communication can be encrypted, for example using TLS (Transport Layer Security). Both devices, the verifier and DMA'n'Play To-Go are trusted.

The third requirement is the knowledge of the firmware of the attested device and its benign states. For this, the verifier needs to access the firmware in the ELF (Executable and Linkable Format) binary format. Note that DMA'n'Play does not necessarily require source code. When compiled with debugging symbols, the verifier can identify the addresses of variables and data in memory by their names in the source code. To check the content of variables and identify illegal states, the verifier requires information about benign states, for example, valid ranges of variables. We provide a configuration file format in which this information can be provided along with integration guidelines that describe how to integrate DMA'n'Play into new or existing applications, see Section 5.5.4.

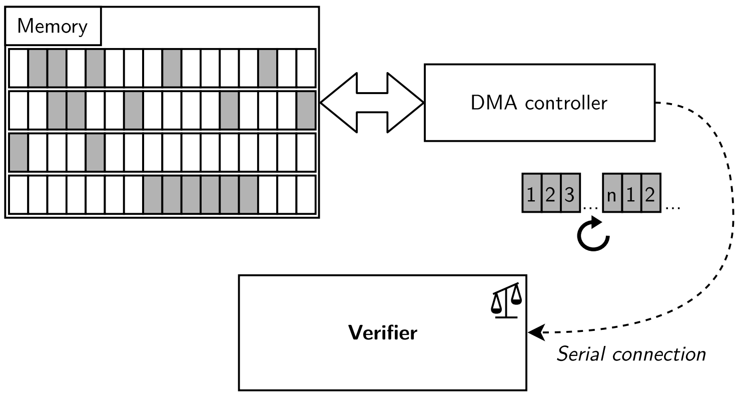


Figure 5.5: The DMA controller sends all relevant memory sections to the verifier in a circular process.

5.3.2 Threat Model

We assume a remote adversary. The adversary can compromise the attested device at any point in time and is able to modify program data or configuration data, for example, by means of typical software vulnerabilities like memory errors or insecure or insufficiently protected interfaces. However, highly privileged operations like changes to the MPU (Memory Protection Unit) are not possible. In Section 5.4.4, we show how this can be achieved on standard commodity microcontrollers.

Similar to other remote attestation approaches, we exclude physical attacks on the devices [3, 4, 201, 241]. Thus, the attested device and particularly its hardware, cannot be tampered with, including the serial connection between the prover and the external verification device. Furthermore, also along with other remote attestation approaches, we assume that attacks on the verifier are out of scope.

5.4 Concept of DMA'n'Play

Our concept of DMA'N'PLAY is to use direct memory access (DMA) to enable an external device, called verifier, to observe the memory of the attested device. Figure 5.5 shows the high-level idea of our approach. In an infinite loop, the DMA controller sends relevant memory content to the verifier, allowing the verifier to monitor memory contents such as configuration data, measurements, and other static and variable memory content. We use a one-way serial connection to send the data of the attested device to the verifier, so there are no interdependencies between these devices. Furthermore, since there is no feedback from the verifier, the attested device cannot determine whether the verifier device is present.

5.4.1 Using DMA for Attestation

For DMA'n'Play, we configure the DMA controller in such a way that it shifts memory contents for attestation to an external peripheral via a serial connection, for example, UART. The verifier receives the raw memory contents from the attested device and verifies its integrity. Serial communications like UART use two separate lines for sending and receiving (see Section 5.1.1). By only connecting the pins for sending on the attested device with the receiving pin of the verifier, a one-way transmission is ensured. This gives two security benefits in contrast to traditional attestation schemes. First, the attested device does not get any feedback from the verifier. Hence, the attested device cannot determine whether it is currently being attested. Second, in case of implementation flaws in the verifier, the attested device faces significant limitations to exploit these flaws as there is no feedback from the verifier.

DMA tasks are usually configured once on set up and are typically not required to change during runtime. We set up the DMA controller to frequently push memory contents (SRAM, configuration data, content of variables) over UART for attestation. Since DMA does not need to be reconfigured, access to the DMA controller can be blocked by the memory protection unit (MPU). Consequently, the external device will always receive untampered memory contents. As the DMA controller is independent of the processor and the software operating on the device, this does not influence the normal operation of the attested system. This makes DMA'n'Play also suitable for attesting devices with real-time constraints, that is, where the correct operation also requires maintaining strict timing thresholds.

5.4.2 DMA'n'Play Attestation

The DMA'n'Play attestation scheme takes advantage of its full memory access to ensure runtime constraints on specific variables. This is unlike traditional attestation schemes, in which memory is being hashed and then sent to the verifier for verification, which makes reconstructing the original content a challenging and complex task. For instance, in these schemes, the verifier compares the hash values to a list of known hash values of benign states, requiring a database of hash values of all valid states.

This approach has several drawbacks. It requires a pre-computation of valid states, an extensive task, leading to the well-known state explosion problem [276]. Changes to the attested device require updates of these states, even upon small changes such as modifications of individual parameters or updates to the attested application. Furthermore, the database of valid states requires significant memory on the verifier's side.

On the contrary, in DMA'n'Play attestation, the verifier has access to the raw data in the memory of the attested device. By mapping the memory content to relevant information like data or control variables, the verifier can monitor the attested device's internal state. With DMA'n'Play, we recommend a model-based approach to avoid the state explosion. That is, the behavior of a model of the attested device is compared to its actual behavior. Direct access to the raw memory content allows the verifier to perform complex checks on state data, for example, assuring that variables are within specific ranges or validating specific dependencies between variables.

To attest a device, the verifier solely requires the binary of the attested device and a configuration file that specifies benign states (see Section 5.5.2). If the attested device is modified or updated, only the binary has to be replaced. If the configuration changes, only the rules in the configuration file have to be adapted. This allows simple updates of the attestation rules, when the attested system changes, for example, due to a new software version.

By mapping the memory content to relevant information like sensor or control variables, the verifier can reconstruct the state and behavior of the attested device. Relations between sensor information and output variables allow detecting compromises and manipulations, creating a deviation between the actual and expected behavior. However, creating such a behavior model is challenging without source code and deep knowledge of the device. Reverse engineering is helpful, but often requires significant effort as states and interdependencies are hard to reconstruct.

However, when the binary is enhanced with debugging symbols or even the full source code is available, the verifier can identify variables and data structures by their names. With DMA'n'Play and application knowledge, it is then also possible to implement sophisticated verification logic and rules for device behavior. The more details about the behavior of the attested device are available, the more details DMA'n'Play is able to monitor. With a full knowledge about the application, also fine-grained monitoring is possible. Furthermore, the verifier can also monitor changes over time or access sensor input. This can be used to validate plausibility of sensor readings or match sensor input with device behavior. In embedded devices, communication with external peripherals, such as sensors, typically takes place through special memory areas, which can also be covered by DMA'n'Play. In addition, the verifier can use sensor information from other attested devices, for example, drones in a swarm, and compare it with the currently attested device.

Summing up, the complete availability of the attested system's memory and unrestricted access to it allow straightforward implementation of checks of the attested system to verify its integrity and detect manipulations. While DMA'n'Play does not necessarily require the source code of the attested system, developing sophisticated rules for the verifier requires insight into the precise functionality of the attested system, which is typically only given via source code.

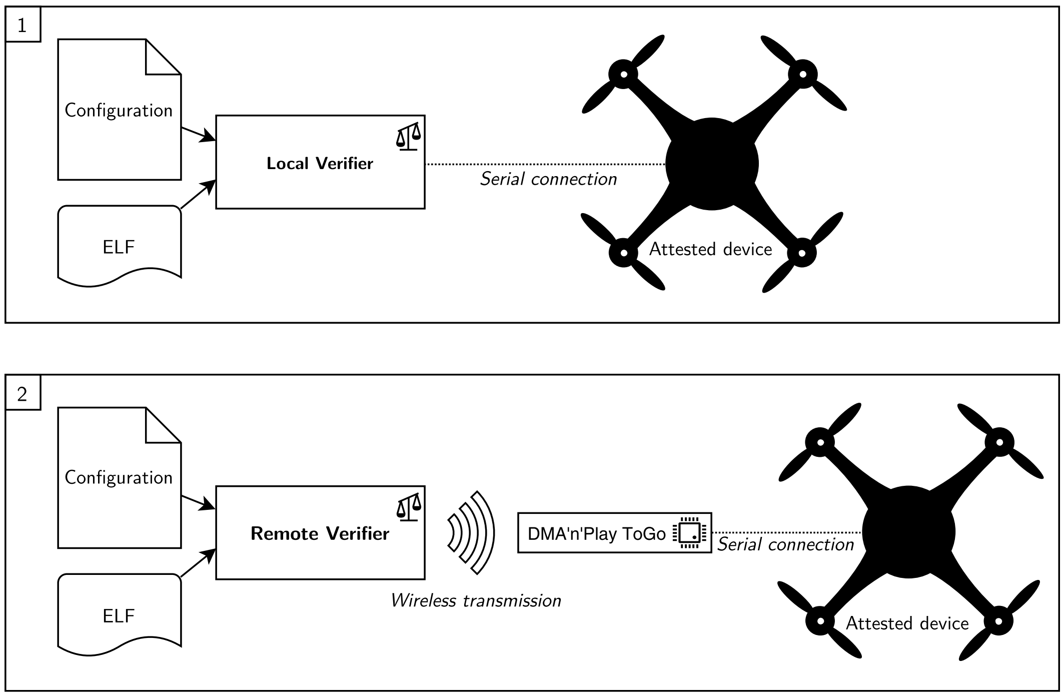


Figure 5.6: The verifier uses the compiled binary and a configuration file to attest the device. The verifier can either be directly connected to the attested device (Case 1), or communicate via DMA'n'Play To-Go (Case 2).

5.4.3 Conception of the DMA'n'Play Verifier

The verifier checks the correctness of the attested device based on the raw memory content data provided via DMA. This data is automatically sent in a circular process by the memory controller of the attested device. For the attestation, the verifier needs to interpret these raw values. To do so, the verifier takes the compiled binary ELF file and analyzes it to obtain the memory regions to be attested. Note that DMA'n'Play does not need the source code of the firmware of the attested device. Embedded devices typically have a static memory configuration. Therefore, the exact memory layout can be initially determined using the binary firmware.

If the binary is compiled with debugging symbols, the verifier can identify variables and data in memory by their respective names, find their location in the data stream, and reconstruct the content of the memory in the attested device. This makes it possible to perform complex checks on the memory of the attested device.

DMA'n'Play requires the verifier device to be close to the attested device due to the communication channel. The verifier can be implemented on any commodity computer system as long as it can be equipped with a serial interface, such as personal computers, mobile devices such as smartphones or tablets, or specialized systems like a diagnostics terminal for cars or planes.

For the attested device, it does not matter whether a verifier is attached or not. As there is no input from the receiving verifier device, the attested device is unable to ascertain whether a verifier is present. Therefore, the attacker cannot determine whether the device is being attested or identify the memory locations that are currently being transmitted. However, an attacker could use other information on the device to determine if it is likely that the device is being attested: For example, in the case of a plane or a car, if it is flying or driving. For instance, it is unlikely that an external verifier integrated in a diagnostics system, that is usually used in a garage, is attached while a car is moving. To counter this, we developed a verification solution called DMA'n'Play To-Go that can be integrated into other devices to continuously attest devices also during operation. DMA'n'Play To-Go is a small embedded device that is being connected to the serial interface and can either directly perform the verifier task or relay the data via a wireless interface, for example, Bluetooth or Wi-Fi, to a remote verifier. In practice, DMA'n'Play To-Go will be used to forward the attestation measurement to an external verifier, as this allows the integration of more complex attestation tasks and also the usage of configuration files for verification, which a small embedded device is not able to process.

Figure 5.6 shows the general architecture of DMA'n'Play and its two operating modes: The verifier can either be directly connected to the attested device (Case 1) or communicate via DMA'n'Play To-Go (Case 2). In the latter case, DMA'n'Play To-Go is directly connected to the attested device and relays the data to the verifier, for example, over Wi-Fi or Bluetooth. The attested device in Figure 5.6 is represented by a drone. In Section 5.6.1 we provide a case study on a syringe pump and a drone.

5.4.4 Locking of DMA Controllers

The DMA controller sends the content of the attested memory to the verifier. The security of DMA'n'Play attestation is based on the assumption that the attacker cannot change the configuration of the DMA controller. Otherwise, the attacker could alter the DMA controller such that critical memory areas are not being reported, thereby hiding modifications and attacks.

Embedded devices in general feature a memory controller, which allows restricting access to arbitrary memory regions. The most basic form of such a controller is the Memory Protection Unit (MPU). It is a common and widespread feature on standard controllers [142, 146, 261]. A properly configured MPU will define protected memory areas and block unprivileged access. It is possible to lock the DMA configuration, by restricting unprivileged access to the memory section that contains the corresponding configuration.

Devices with an MPU should provide at least two privilege modes. The basic configuration has one privileged mode with access to all resources, and one unprivileged mode with limited capability. In order to ensure the memory access rights, the code is run either in privileged or unprivileged mode. Within the unprivileged mode, all memory restrictions defined in the MPU are strictly enforced. Once the processor operates in unprivileged mode, switching back to privileged access is only possible through a Super Visor Call (SVC). This triggers an interrupt that checks the legitimacy of the request and either allows or denies the mode switch. Since MPU restrictions are only enforced in unprivileged modes, it is important to avoid critical bugs in privileged code. Therefore, firmware analysis [181] and fuzzing [110] are performed. Furthermore, the amount of privileged code is minimized by separating tasks based on their required permission level. This is often implemented in software, for instance, by the operating system of the microcontroller. TockOS [173], for example, divides the OS kernel into a trusted core for critical tasks and untrusted capsules for peripheral drivers and other noncritical tasks. EPOXY [70] uses the MPU to provide two domains and requires manual annotations by the developer. Sometimes ISA properties, such as unprivileged memory instructions, are used to enable execute-only memory protection schemes (uXOM [167]).

While privilege separation has been neglected in the past it is now a critical task for the software developer. Recently, frameworks such as EPOXY [70] have emerged in academic research, applying a technique called privilege overlaying to only execute the necessary operations in privileged mode. This considerably advances the protection of hardware configurations including the DMA controller. Some frameworks such as D-Box [180] explicitly address the topic of DMA locking and DMA security. D-Box [180] allows the compartmentalization of the DMA controller on embedded devices like ARMv7-M boards, using a software reference architecture and the capability of the MPU. Thus, the DMA configuration can be sufficiently protected from a potential attacker. EPOXY [70] and D-Box [180] enable us to provide a secure channel over DMA to the external verifier. Thus, we are able to add remote attestation to devices, that had no feasible attestation option until now (either due to hardware or system limitations). Some microcontrollers even feature more sophisticated protection solutions, such as a complete memory management unit (MMU), for example, the ARM Cortex-A family [31], up to a full trusted execution environment (TEE) such as TrustZone. TEEs guarantee authenticity of the executed code, integrity of runtime states, and confidentiality of code and data [228]. For example, TrustZone is an optional, but common extension of the new and more sophisticated ARMv8-M [27] and ARMv8-A [28] microcontroller architecture. Unfortunately, as of now, TrustZone has very limited availability on existing microcontrollers [180]. In the RISC-V architecture such memory restrictions are enabled using the Physical Memory Protection (PMP) features included in the RISC-V instruction set [222]. Furthermore, for RISC-V there exist TEE solutions such as Keystone [171]. However, as of now, Keystone requires the privilege modes S, U, and M [171]. Unfortunately, these privilege modes are also optional and hence often not implemented. For example, the new and popular ESP32-C3 only implements the unprivileged U (user) and privileged M (machine) mode [104].

5.4.5 Hardware Requirements & Target Platforms

In summary, DMA'n'Play requires three hardware properties:

1. A DMA controller with a peripheral such as SPI or UART to directly output memory contents to the external attestation device.
2. An MPU, which locks the DMA configuration.
3. Privilege separation with at least two modes to prevent reconfiguration of the DMA controller by the attacker.

Numerous hardware platforms fulfil these requirements. We focus on the ARMv7-M architecture, which we also use in our case study in Section 5.6.1. This architecture is in widespread use in the industry with many legacy devices. The successor ARMv8-M is also suitable for DMA'n'Play. It even has optional support for TrustZone [228]. However, TrustZone support is purely optional and there will be new boards without TEE. Recently, the RISC-V architecture became popular, especially in the embedded domain. As discussed in Section 5.4.4, DMA'n'Play can also be used on RISC-V devices.

5.4.6 Devices Without Source Code

The DMA'n'Play framework can also be used on applications where there is no source code, but only the compiled binary is available. For example, the source code of legacy devices is often either missing, incomplete, or unavailable. In particular, there is a huge amount of legacy devices in machinery designed for a long service life, for example, in power plants, factories, professional equipment, and vehicles. In these environments, there are often embedded devices with discontinued software support, leading to critical security risks [161, 218]. If no source code is available, DMA'n'Play can be integrated with the help of binary rewriting. Binary rewriting describes the modification of a given compiled and possibly (dynamically) linked binary in such a way that it remains executable [283]. Binary rewriting can either be done dynamically (during execution) or statically (on a binary that is not currently being executed) [283]. Due to the added complexity of runtime execution, dynamic rewriting is more challenging than static rewriting. In static rewriting, all binary transformation steps can be executed in a row, while dynamic rewriting requires an iterative algorithm [283]. Furthermore, a persistent dynamic binary transformation will induce time and memory overhead during runtime [283]. For most IoT devices static rewriting will be sufficient to integrate DMA'n'Play, as devices can be usually re-flashed with a new binary during maintenance.

In order to integrate DMA'n'Play into a binary the following steps are required:

1. Integrate a DMA configuration into the binary, to output the entire memory through DMA.
2. Integrate an MPU-based lock.
3. Set up the verifier with information on the memory content and data of interest for the attestation process.

Depending on the amount of information available regarding the memory structure and the variables of interest, the verifier can be configured appropriately. The integration guidelines described in Section 5.5.4 apply. The only difference is that the DMA output and MPU lock on the attested device are integrated and configured with the help of binary rewriting techniques, rather than directly added to the source code and recompiled.

The integration of the DMA output (Step 1) and MPU-based locking (Step 2) are straightforward on the binary level. Both steps represent a reconfiguration of the hardware. On the software side, this reconfiguration is equivalent to privileged write operations on registers and memory positions (Memory Mapped I/O [221]). DMA'n'Play performs this configuration step once during device startup and drops all privileges afterward. Thus, steps 1 and 2 can be achieved by adding a static code block to the part of the binary executed at the end of the device startup.

5.5 Implementation

To show the applicability of DMA'n'Play for different computing architectures, we integrated DMA'n'Play into a syringe pump and a drone. A syringe pump is a medical device that automatically injects medicine into a patient's body. Drones are manually controlled or autonomous flying devices. First, the attested device (syringe pump or drone) must be modified to send its memory contents to the verifier. Second, the verifier has to be provided with information on the benign states of the attested device. Next, the verifier needs to check the validity of the received data and report manipulations. In Section 5.5.4 we provide detailed integration guidelines that describe how DMA'n'Play can be integrated into devices.

5.5.1 Attested Device

To implement DMA'n'Play on the attested device, we first determine the relevant memory content. These memory contents typically include variables that represent

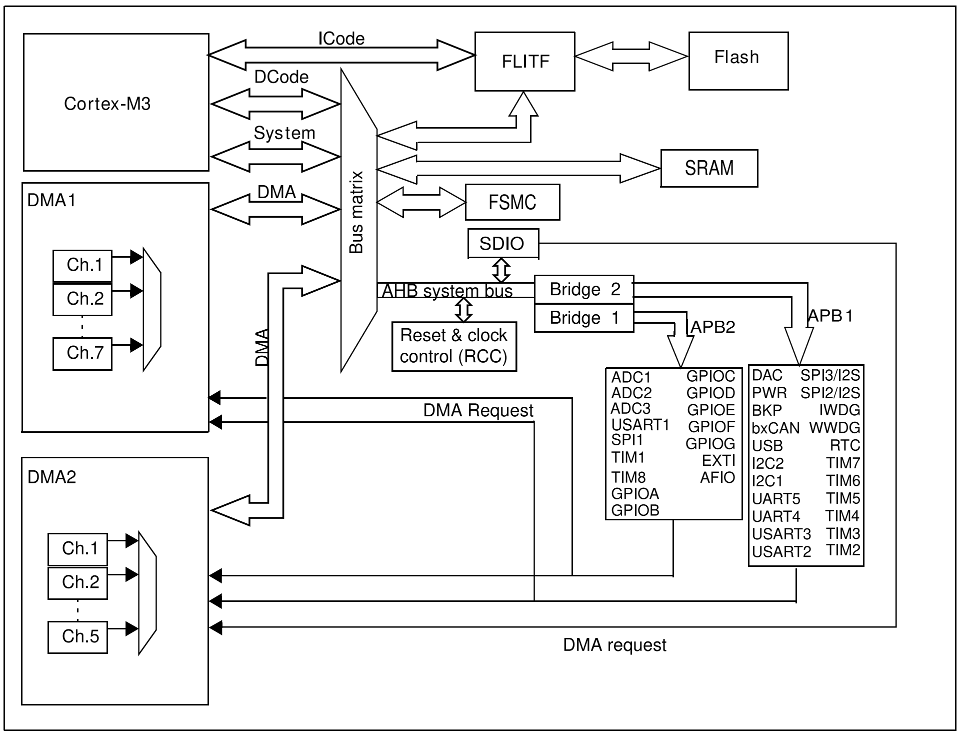

ICode
DCode
System
DMA
Cortex-M3
FLITF
Flash
SRAM
FSMC
SDIO
Bus matrix
AHB system bus
Bridge 2
Bridge 1
APB2
APB1
Reset & clock control (RCC)
DMA1
Ch.1
Ch.2
Ch.7
DMA2
Ch.1
Ch.2
Ch.5
DMA
DMA Request
DMA request
ADC1 ADC2 ADC3 USART1 SPI1 TIM1 TIM8 GPIOA GPIOB
GPIOC GPIOD GPIOE GPIOF GPIOG EXTI AFIO
DAC PWR BKP bxCAN USB I2C2 I2C1 UART5 UART4 USART3 USART2
SPI3/I2S SPI2/I2S IWDG WWDG RTC TIM7 TIM6 TIM5 TIM4 TIM3 TIM2

Figure 5.7: The ARMv7 architecture features two DMA controllers [262].

the state and configuration of the device. To reduce the amount of transmitted data, we modify the linker file to create a dedicated memory section containing all the data to be included in the attestation. We call this section the 'attestation section'. This is an optional step, it is also possible to attest the complete memory of the attested device. We use the built-in source code annotation capability of the gcc compiler, called attributes [124], to assign variables to the 'attestation section'. Such functionality is also available in other compilers such as clang [271].

In Section 5.4.1, we briefly describe the DMA feature that we use to transfer memory content to an external device. As shown in Figure 5.7, the DMA controller is independent of the main CPU and has direct access to memory as well as peripherals such as SPI, UART, or I2C. We configure the DMA controller on the chip to output the contents of the attestation section over a peripheral interface. We set the DMA controller to circular mode and configure a direct DMA stream from memory to peripheral. Thus, the attested device sends the memory content in an endless loop.

To enable the verifier to determine the current position in memory, the attested memory features a so-called attestation header, which is a static string at a known position in the attested memory. The attestation header is placed in the attested

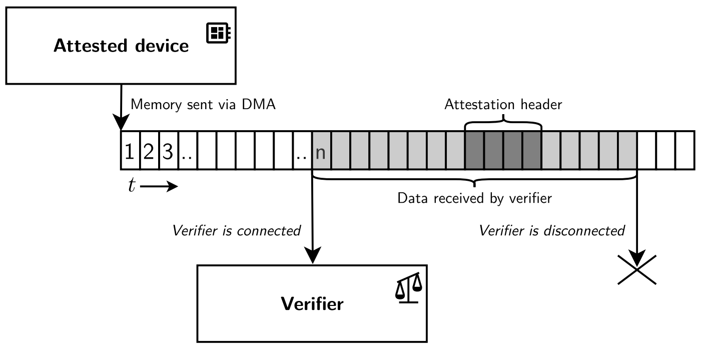


Figure 5.8: The verifier uses the attestation header to identify the position in the data stream.

memory while developing the verifier rules. As embedded devices typically have a static memory configuration, this attestation header has a fixed position in the memory that does not change during runtime and also remains at the same position during restarts of the device. The content of all variables and data structures in the attestation section can be identified using the attestation header as a reference point. This is especially relevant as the verifier can be attached at any time. Thus, the data stream of the attested device can be at any random position. Figure 5.8 illustrates this scenario.

To secure the DMA controller from malicious access, we utilize the Memory Protection Unit (MPU) as described in Section 5.4.4. Since the configuration of the DMA is handled through memory-mapped I/O, we restrict access to the memory area that contains the DMA configuration. We drop the privilege level that is required to reconfigure DMA after startup. Thus, a remote attacker cannot alter or influence the data transmitted to the external verifier via DMA. Even if the remote attacker gains arbitrary code execution on the device, the attacker still misses the required privilege level to alter the DMA configuration.

Typically, the source code of the attested device is available and can be recompiled with our modifications, that is, the dedicated attestation memory section, modified DMA configuration, and MPU-based lock. However, source code is unavailable for some legacy devices for various reasons. In case the source code is not available, we output the entire memory content for attestation and leverage binary rewriting techniques. We provide further details on this in Section 5.4.6. In the integration guidelines in Section 5.5.4 we explain all the steps necessary to integrate DMA'n'Play.

5.5.2 Verifier

The verifier receives and validates the data from the attested device. To do so, the verifier requires an ELF file, that is, the binary of the software running on the attested device. With this file, the verifier automatically reconstructs the memory layout of the attested device, to determine the locations of variables and identify data structures. Using this information, the verifier determines the location and values of these variables in the data stream from the attested device. When compiled with debugging symbols, the verifier can identify the addresses of variables and data in memory by their names in the source code. Otherwise, manual mapping of variable names to memory addresses is required. Embedded devices typically feature a static memory configuration. So, this memory layout does not change during runtime.

We implemented the verifier in Python with pySerial[1] for serial communication and the pyelftools[2] to handle the ELF file. There are two possibilities for the verifier to validate the received information. First, the configuration file format can be used to define constraints and rules that are checked. This allows for straightforward implementation of the verifier. Alternatively, complex checks can be manually implemented into the DMA'n'Play framework.

In a configuration file, the developer can provide constraints for these variables. With this configuration file and the ELF file of the attested device, the verifier automatically checks the received data stream. No further manual implementation steps are required. The format of the configuration is simple: Variables are identified by their name in the source code. Furthermore, the developer has to provide the variable type and the constraints to be checked. Listing 5.1 shows an example of such a configuration that describes the data structure of the attested memory. In Listing 5.2, restrictions and logical constraints are defined.

DMA'n'Play supports the following checks:

- Static checks that have to be fulfilled, i.e., values that may not change.
- Lists of alternatives i.e., different valid values.
- Arbitrary values, i.e., any value is valid.
- Ranges, i.e., specific ranges of valid values.

This functionality can be flexibly adapted to further use cases and checks. Alternatively, complex checks can also be implemented. As the verifier is developed in Python, adding further checks or developing more complex rules is simple. Even integration into other systems, such as back-end and management systems or web services is possible. In contrast to traditional attestation schemes, no preliminary exploration of possible states is required, easing implementation and modifications to existing systems. In remote attestation schemes that use a precomputed list of

[1]`https://github.com/pyserial/pyserial`
[2]`https://github.com/eliben/pyelftools`

Listing 5.1: Example configuration file for verifier

```
layout = cstruct.Struct(
 "p_settings"         / p_settings,
 "dosage_ml"          / cstruct.Int16ul,
 "bolus_steps_ml"     / cstruct.Array(9, cstruct.
    Float32l),
 "attestation_header" / cstruct.Array(3, cstruct.Int8ul)
)
```

Listing 5.2: Example for valid ranges of variables for verifier

```
varmap = {
  "p_settings:syringe_volume_ml":(DataModel.VarType.
    STATIC,DataModel.VarStatic(30)),
  "dosage_ml":(DataModel.VarType.RANGE,DataModel.
    VarRange(0,6)),
  "bolus_step_index":(DataModel.VarType.ALTERNATIVES,
    DataModel.VarAlternatives([0,1,2,3,4,5,6,7,8])),
  "attestation_header" : (DataModel.VarType.ANY,
    DataModel.VarAny(None))
}
```

valid states as hashes, this list has to be recomputed on every change of the attested system. The usage of a memory-safe language like Python for the verifier prevents a compromise of the verifier due to memory corruption. Note that the input the attested device provides should be considered untrusted during development.

5.5.3 DMA'n'Play To-Go

The verifier is an external device that has to be directly connected to the attested device. To compensate for limitations induced by this design, we developed an external device called DMA'n'Play To-Go that can relay attestation data to a different verifier. DMA'n'Play To-Go is an embedded device with a small form factor, little power consumption, and is available at a low price so that it can be integrated into the attested systems. In particular, we used an ATmega328P 8-Bit microcontroller with 16 MHz clock frequency and 2 kbyte of SRAM [35]. It consumes 2 to 10 mA at full load depending on frequency [184].

Figure 5.9 shows a picture of our prototype implementation. The gray cable connects the attested device with the receiving pin (RX) on the microcontroller. The transmission pin (TX) of the microcontroller remains unconnected. A one-way

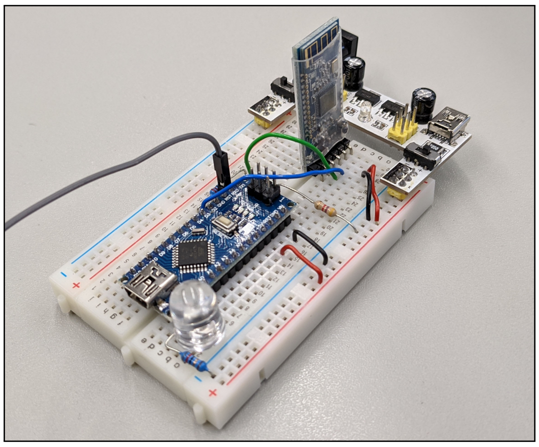


Figure 5.9: Photo of the DMA'n'Play To-Go prototype.

connection is ensured by only connecting the attested device with the receiving pin (RX) on the microcontroller, leaving the transmission pin (TX) of the microcontroller unconnected.

Our implementation used Bluetooth Low Energy (LE) to transmit the data stream. Bluetooth LE is well-suited for this use case as it has a low power consumption and a reach of up to 100 meters. Alternatively, also a microcontroller with Wi-Fi functionality such as the ESP8266 [98] or the ESP32 [97] can be used to send the attestation data to a remote server or cloud service. Alternatively, these microcontrollers can also be used as a verifier, even though they do not have enough computational capabilities to analyze the binary of the attested device. Therefore, the verification logic has to be manually implemented on these microcontrollers.

5.5.4 Integration Guidelines

In this section, we show the steps required to integrate DMA'n'Play into a new application and set up the corresponding verifier.

1. Identify the memory areas to be attested. When compiling the attested application, all relevant memory areas can be moved into a dedicated memory section. In case no source code is available, also the complete memory can be attested.

2. Integrate an attestation header, that is, a unique, identifiable string of bits, into the attested memory, for example, the attested memory section. Note that the attestation header does not need to be placed in a specific position as long as it is inside the attested memory region. In case no source code is available,

choose an existing and unique bit string within the attested memory to serve as the attestation header.

3. Configure the DMA controller to output the attested memory to the bus used for the attestation, for example, the serial bus. Take care of the respective bus configuration, for example, transmission speeds.
4. Define valid states of the attested device and develop a configuration file containing a rule set. When using a binary with debug symbols, the verifier can identify variables by their names. For more information on how to build a configuration file, see Section 5.5.2.
5. Set up the verifier. This includes configuring the bus used for the attestation and providing the binary of the attested device along with the configuration file.

These steps allow integrating DMA'n'Play into new as well as legacy devices. Note that no source code of the attested device is required. Changing the configuration or updating the binary requires only a subset of these steps, for example, replacing the binary, or modifying or updating rules in the configuration file.

5.6 Evaluation

We implement DMA'n'Play into the syringe pump and a drone and attest numerous variables of different types to show the capabilities of the verifier. For each device, we design and execute a typical practical attack to show that DMA'n'Play is able to detect the compromise of configuration data. We performed timing measurements to show that DMA'n'Play has no timing influence on normal operation. This shows that DMA'n'Play is even suitable for systems with real-time constraints.

5.6.1 End-to-End Case Study

For the evaluation, we integrated the DMA'n'Play attestation into two real-world devices, a syringe pump, and a drone, using our integration guidelines. Both devices perform safety-critical tasks and operate under real-time constraints. We then developed full end-to-end examples and integrated a typical vulnerability in each device. Next, we created a configuration file, defining valid states and ranges for both devices. Upon exploiting the vulnerabilities and applying the respective attack, these were immediately detected. There are different methods to respond to a detected compromise. In traditional attestation schemes the verifier has no influence on the attested device, and can for instance solely raise an alarm. In contrast, in the case of DMA'n'Play, the verifier can also interact directly with the attested device.

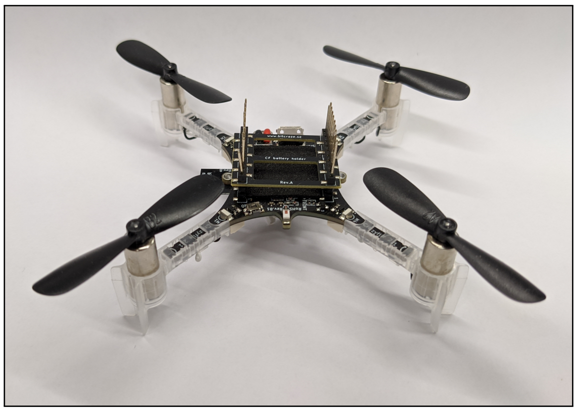


Figure 5.10: Photo of the Bitcraze Crazyflie 2.1 drone.

For example, the verifier could power off the syringe and alert a doctor in case of a compromised syringe pump. In the case of an autonomous drone, the compromised drone could be excluded from an autonomous swarm, shut down, or set into a secure configuration. In any case, this requires an adequate connection between the verifier and the attested device.

Syringe Pump. A syringe pump is a medical device that injects medicine into the body of a patient. We enhanced an open-source syringe pump [285] with DMA'n'Play and implemented it using a Nucleo-F446RE development board that features an ARM Cortex-M4 processor. DMA'n'Play continuously monitors the devices' configuration so that any illegal operations are detected. To do so, we wrote the corresponding configurations for DMA'n'Play. Then, we integrated a common vulnerability into the syringe pump: an insecure configuration interface. This configuration interface allows changing the amount of medicine being injected. The attack can have potentially lethal consequences for the patient if too much or too little medicine is being injected. As stated, such an attack vector is typical for IoT devices: the most common vulnerabilities are weak, guessable, or hard-coded passwords, as well as insecure network interfaces and services [208].

Drone. We integrated DMA'n'Play into a Bitcraze Crazyflie 2.1 drone [41]. Figure 5.10 shows a photo of the drone. Drones feature many safety-critical components that are crucial for correct operation. A malfunction or compromise of one of these components can have severe consequences. For the attack, we use the remote control channel of the drone that is used to send new commands and fly the drone directly. For a remote adversary, this is the primary attack vector. Using this channel, we compromised the device and changed multiple critical values: The configuration of the state estimator determines the flight position and stabilization of the drone. This critical component processes the sensor data from the drone and provides the position and movement of the drone. Changing the parameters of the estimator directly influences the flight behavior of the drone. Moreover, as the control and navigational system of the drone are dependent on this system, these modifications allow controlling the drone without directly interfering with its navigational system. In the case of autonomous drones, this attack will be undetected as the autonomous control system remains unmodified. However, this is only an example of an attack: any flight parameters or configurations could be manipulated, resulting in arbitrary attacker-controlled behavior. It would also be possible to directly control the drone by altering its navigational and way-finding system, or compromising the collision warning system so that the drone collides and crashes.

We integrated the DMA'n'Play framework into both devices and configured it to monitor critical components and data. To do so, we defined valid states and ranges for variables in a configuration file. We provided this configuration file to the verifier together with the binary of the software running on the attested device. Then, we monitored the normal operation of the device. Upon exploiting the vulnerabilities and applying the respective attack, the verifier immediately detected these by raising an alarm. In practice, there are different methods to respond to a detected compromise, for example, by raising an alert, or, in autonomous settings with multiple devices, isolating a compromised device. As mentioned before, in the case of DMA'n'Play, the verifier can also interact directly with the attested device, for example by powering it off in the case of the syringe pump or excluding a drone from a swarm in the case of autonomous drones.

5.6.2 Real-Time Capabilities

By using DMA and an external verification device, which are independent of the processor's normal operation, DMA'n'Play can even be applied to hard real-time applications. Such systems have strict responsiveness requirements that limit the integration of new security techniques.

The data transmission on the DMA controller does not consume any CPU time, because DMA is a dedicated hardware unit with its own access to the memory bus and peripheral interfaces. In general, the CPU and the DMA controller both act as master devices on the memory bus [258, 260]. Since the DMA and the CPU could potentially compete for the memory bus usage, round-robin arbitration mechanisms are implemented in hardware [258, 260]. Memory buses can be optimized for either bandwidth or low latency in sharing. For microcontrollers, such as the ARM Cortex-M family, memory buses are optimized for low latency in sharing [260].

Low sharing latency means that very fast switches occur between memory access tasks. In addition, DMA latencies can be precisely predicted [258]. Since peripheral interfaces are relatively slow compared to the speed of the memory bus, the additional operations on the memory bus are limited. Furthermore, due to the low latency sharing (round-robin arbitration on the memory bus) implemented between CPU and DMA, as well as the relatively low speed of peripheral interfaces, there is negligible impact on the operations performed by the processor.

5.6.3 System Performance View

The attestation mechanism via DMA can potentially compete for memory bandwidth with the CPU. As mentioned in Section 5.6.2, the hardware implements round-robin arbitration: Thus, DMA can only occupy up to 50% of the memory bandwidth, but would not starve the CPU on memory accesses. The overall utilization of the memory bandwidth varies from application to application. However, the memory usage of DMA'n'Play will be far below the maximum bandwidth usage of 50% as the transmission speeds of typical peripheral buses such as UART or SPI are much less than the memory bus speed. Figure 5.12 shows the transmission speed of standard UART and SPI configurations.

A typical UART data rate of 115,200 baud is approximately 11.25 kbyte/s, and an SPI connection clocked with 40 MHz can transfer up to 5000 kbyte/s. The total capacity of the memory bus is much higher than those achievable over peripherals such as UART and SPI. The Cortex-M4 core used in our case study in Section 5.6.1 features a 32-bit AHB Lite Bus for the memory interface [29]. According to the bus specification [23] the transfer consists of one address cycle and at least one or multiple data cycles. Due to the 32-bit bus width, 4 byte (32 bit) can be transferred per cycle.

In contrast, SPI is only capable of a single-bit transfer per clock cycle. Therefore, the SPI peripheral is usually not clocked at full processor speed, but rather at $\frac{1}{2}$ or $\frac{1}{4}$ of the processor clock. Assuming the drone use case from Section 5.6.1 we used a processor clock of 160 MHz on the Cortex-M4, the AHB bus clocked at 160 MHz is capable of transmitting 640 Mbyte/s. The 40 MHz SPI connection (5 Mbyte) would only take $\frac{1}{128}$ and UART at 115,200 baud would only take $\frac{1}{56888}$ of the available memory bandwidth. Therefore, the memory impact of DMA'n'Play is limited and can be configured by selecting adequate transmission speeds.

5.6.4 Feasibility of Full Memory Attestation

Full memory attestation is feasible, but the attested memory portion and transfer speed always need to be carefully chosen to avoid the well-known time-of-check/time-of-use problem (TOCTOU, see also Section 5.7). Attesting 128 kbyte of memory with

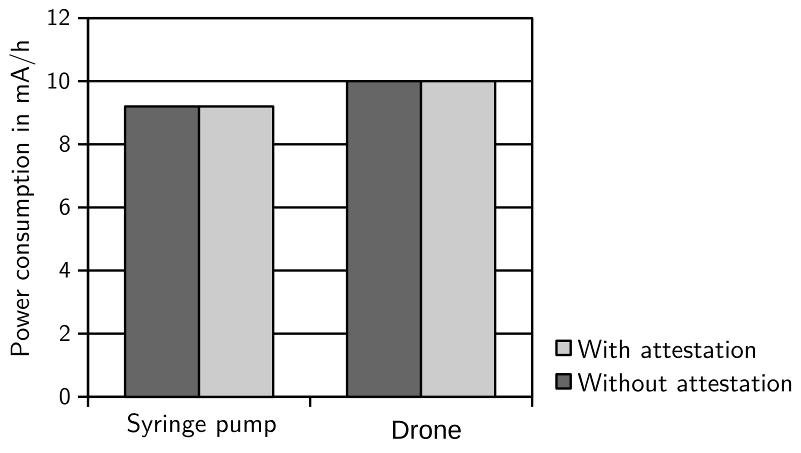


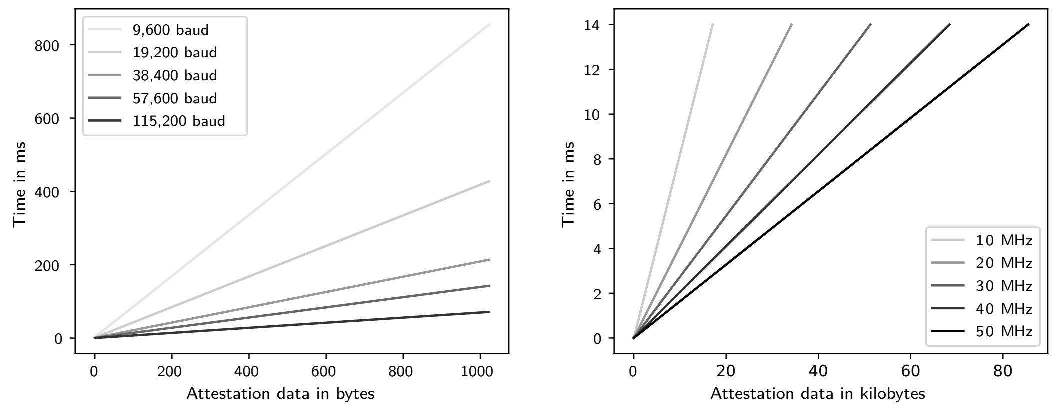
Figure 5.11: Power consumption of DMA'n'Play in the case study.

40 MHz SPI takes about 25 ms. On embedded flash memory chips, there is typically a huge speed difference between reads, which are fast, and writes, which are very slow. Since flash memory has to be written sector-wise, to write any data, at least one entire sector must first be erased and re-written. Erasing a sector of a common chip requires around 50 ms [93]. Slower flash chips even take up to 100 ms [92]. In practice, this circumstance eases attestation as it effectively mitigates the TOCTOU problem.

5.6.5 Power Consumption

Power consumption is an important aspect, especially in embedded systems, which are often battery-operated. We measured the power consumption of our two prototype implementations for 7 min and measured the total energy consumption. Figure 5.11 shows the results. The drone consumed 50 mW h, the syringe pump 46 mW h, adding up to 10 mA h and 9.2 mA h respectively. In summary, we could not measure an increase in the power consumption of the devices running DMA'n'Play compared to the default implementation. Hence, DMA'n'Play is suitable for mobile applications and small embedded devices. However, integration of DMA'n'Play can slightly increase the power consumption of a system due to the influence on the deep-sleep behavior of the processor [270].

Although the integration of DMA'n'Play does not influence the power consumption of the attested device, the verifier also requires power. Depending on the processor frequency DMA'n'Play To-Go consumes between 2 and 10 mA at full load [184]. While in absolute numbers this seems low, depending on the attested device, the power consumption can be significant. However, the actual power con-



(a) UART. (b) SPI.

Figure 5.12: Trade-off between bus speed, amount of attested data, and transmission time.

sumption of the verifier depends on the actual implementation, the microcontroller, and transmission technologies.

5.6.6 Summary

This evaluation showed the applicability of DMA'n'Play using two real-world examples, a syringe pump, and a drone. In our practical evaluation, we showed how DMA'n'Play is able to detect attacks. Furthermore, we showed that DMA'n'Play does neither increase power consumption nor influence the runtime behavior of the attested device. This allows a wide usage of DMA'n'Play, including devices with real-time constraints.

5.7 Security Discussion

The novel approach of DMA'n'Play to remote attestation has several advantages in practicability compared to traditional remote attestation schemes. But several security aspects have to be considered when using DMA'n'Play.

Attack Model. For a successful attack, the adversary has to modify memory content, such as variables, without being detected. The DMA controller, which is independent of the software running on the attested device, sends the memory content to the external verifier in a circular process. The verifier continuously monitors this memory content. As discussed in Section 5.3, the attacker cannot influence which memory is being monitored, interrupt this process, or falsify the reported data. Therefore, it is crucial that the adversary cannot change or influence the configuration of the DMA controller. In the following, we discuss possible attacks on attestation and show how DMA'n'Play circumvents these.

Time-of-Check/Time-of-Use. The time-of-check/time-of-use problem (TOCTOU) describes the circumstance that in remote attestation schemes, there is a delay between the attestation time when the integrity of the device is checked and execution later on. An attacker can exploit this time span to carry out an attack without detection. Also, DMA'n'Play suffers from this problem: The memory is copied in a circular process to the verifier, leaving a time span between each time a specific memory location is copied. The length of this interval is dependent on two variables: The amount of data that is being verified and the bus transmission speed. Figure 5.12 shows this dependency for typical bus speeds of serial interfaces. The faster the transmission speed, the shorter the interval between two attestation runs. Note that the graph does not start at 0 byte. For DMA'n'Play an attestation header is required to identify the memory layout in the data stream. So, the attestation header has to be included in all transmissions, even the smallest ones. In our implementation, this attestation header takes 3 byte. While this time span between two attestation runs can be exploited in theory, the attacker does not know which memory parts are currently being attested in practice. As eluded in Section 5.4 the transmission of the memory content to the verifier is performed by the DMA controller, which is completely independent of the processor and the software running on it.

Device Authenticity and Offloading. The DMA'n'Play device is physically connected to the attested device, therefore the attacker is not able to trick DMA'n'Play into attesting a different device (offloading). In order to change the input data to the DMA'n'Play device, the attacker has to reconfigure the DMA controller. However, such highly privileged operations like changes to the MPU (Memory Protection Unit) are not possible according to our threat model in Section 5.3. We show how this can be achieved on standard commodity microcontrollers in Section 5.4.4.

Limitations. DMA'n'Play does only detect deviations from predefined behavior. Therefore, modifications to static code in flash will not be discovered. To ensure integrity of static code, secure boot mechanisms can be implemented. Secure boot mechanisms are widely available and often already used in many microcontrollers.

Presence of Attestation. The attested device and the verifier are connected via a one-way serial connection, so there is no feedback channel from the verifier to the attested device. To change this, a new physical connection between the attested device and the verifier has to be established, a task that has to be performed manually. According to our threat model, physical attacks are out of scope. Therefore, the attested device cannot get any response from the verifier. It makes no difference to the attested device whether a verifier is connected or not. Thus, the attested device cannot check whether it is currently being attested.

Side-Channels. However, the attested device could use heuristics to determine if it is attested: Due to the necessity for an external verifier device, in many application areas, an attacker can use side-channels or heuristics to determine whether it is likely that a verifier is present. For example, in the case of a vehicle, the integrity is probably being checked in a garage during maintenance. A drone is unlikely to be attested during flight with a large verifier. Therefore, we developed DMA'n'Play To-Go, a small, embedded device that can be used as an external verifier and be integrated in case a larger external verifier cannot be used. More details on DMA'n'Play To-Go can be found in Sections 5.4.3 and 5.5.3. So the attestation can also run during normal operation, identifying attackers during runtime. Attackers cannot use side-channels to predict whether the device is being attested and which sections are currently being attested.

Security of DMA Controllers. The security of DMA'n'Play attestation is based on the assumption that the attacker cannot alter the data that the DMA controller sends as eluded in Section 5.4.4. Therefore, we must ensure the integrity of the DMA controller's configuration. Protection mechanisms for DMA are different with respect to the targeted platform, and the use case of DMA. DMA is used for fast communication across peripherals such as network and graphic cards in servers and workstations, usually over PCI(E). Thus, servers and workstations feature specific protections like the input-output memory management unit (IOMMU) [5, 15]. However, such protections are not present on MCUs. MCUs have different architectures and requirements. They are used for embedded applications (for example, vehicles such as cars and trains, industrial facilities, or IoT deployments) and require lower communication speeds between peripherals than servers or workstations. The DMA controller in embedded contexts unburdens the CPU from wasting scarce CPU time on managing data transfers, such as UART or SPI data transmissions. For example, a heavy CPU load can limit the system's ability to meet scheduled deadlines, which is important in the embedded context. We extensively elaborate on how to securely lock DMA controllers in Section 5.4.4. We also summarize all requirements and targeted platforms for DMA'n'Play in Section 5.4.5.

Manipulations of the Attestation Header. The attestation header is a crucial component in identifying the components of the attested memory region. This attestation header is controllable by the attacker. This means the attacker can fully manipulate and shift the attestation header. However, the attacker cannot change the location and amount of the memory being attested, as this requires modifying the configuration of the DMA controller. If the attestation header is changed, the verifier does not recognize it, resulting in a failed attestation. If the attestation header is moved, this is detected during attestation as the positions in the data stream change. Furthermore, changes to the position of the attestation header cause a misalignment of the attested data, which also causes the attestation to fail.

Attacks on the Verifier. The security of the verifier is crucial for DMA'n'Play attestation. Although attacks on the verifier are out of scope, we will briefly discuss the security aspects of the verifier. Successful attacks from the attested device to the verifier are unlikely: The attested device and verifier are connected using a one-way serial connection as eluded before. Interrupting this connection or sending modified content will make the attestation fail. The verifier receives a known amount of memory content at a constant rate in a circular process. In this constant process, no complex data structures have to be parsed, and no new memory areas have to be allocated. This makes runtime vulnerabilities caused by typical memory errors highly improbable. The verification process consists of simple comparisons against known information. As the data rate of the serial connection is fixed at a constant rate, denial-of-service (DoS) attacks that jam the verifier are impossible. As explained in Section 5.4 there exists no feedback channel from the DMA'n'Play To-Go to the attested device. Hence, the attacker cannot exfiltrate any information from the verifier, such as, cryptographic keys or configurations. In total, there is no realistic attack scenario on the verifier.

Security of DMA'n'Play To-Go. DMA'n'Play To-Go is a low-end embedded device that receives data from the attested device and relays it to the external verifier. This simple process offers little to no attack surface as the input data is not processed. The fixed transmission rate of the serial connection prevents denial-of-service attacks (DoS) on DMA'n'Play To-Go. If the attacker increases the transmission speed unilaterally, not only will data be incorrectly received by DMA'n'Play To-Go, but the amount of data received will also stay the same. Similar to the verifier, no feedback channel exists from the DMA'n'Play To-Go to the attested device, making data exfiltration impossible.

5.8 Related Work: DMA Security

In the related work, we investigate the security of DMA, as this is a crucial component of DMA'n'Play attestation.

DMA allows direct access to memory by external devices without involving the main processor, thereby increasing the overall system performance. On the downside, this also facilitates a wide range of attacks, compromising the integrity of a system or reading critical parts of main memory [177]. Well-known examples are attacks via Firewire [38, 86], PCIe [117], and Thunderbolt [227]. This attack vector can, for example, be used to obtain the key of the full disk encryption [43, 86]. It is particularly critical in case of external interfaces, such as Firewire and Thunderbolt, which attackers can easily access when they have physical access to the device.

DMA controllers can also host malware [257]. Furthermore, external devices can be untrusted or contain bugs. To counter attacks with external peripheral devices, IOMMU has been introduced [5, 15, 26], enabling memory management for such external devices. However, this has not been proved sufficient [177]. To find vulnerabilities in devices connected via DMA, both, fuzzing [249] and static analysis methods [36] have been proposed.

In addition to direct exploitation, DMA access can also be used for indirect attacks. Network interfaces are a worthwhile target as they enable remote access. Research found that DMA enables remote Rowhammer-style attacks [268], which, in contrast to normal Rowhammer attacks, do not need local code execution on the victim [156]. On Intel processors, network cards can even manipulate and observe the processor's last level cache (LLC), allowing remote Prime+Probe [175] attacks to leak critical information [165].

Recent research found side-channel attacks using DMA [45]. An untrusted DMA device was used to measure subtle timing differences on the shared DMA bus. These timing differences allowed to leak behavior, such as executed instruction, of trusted environments such as VRASED [201] and SANCUS [198, 199]. Both systems have also been used to implement remote attestation [83, 201, 202, 203]. To prevent such attacks, MicroProfiler has been proposed [46]. MicroProfiler allows the identification of the exact timing behavior of all instructions by systematically profiling all instructions of an instruction set architecture (ISA). This information is then used to verify whether a particular program leaks information over side-channels.

5.9 Summary and Conclusions

In this chapter, we presented the DMA'n'Play framework, that leverages direct memory access (DMA) to directly monitor the memory of the attested device. Instead of using trusted hardware components or complex software-based attestation schemes, we leverage DMA to give a tiny embedded device access to the memory.

In contrast to traditional remote attestation schemes, DMA'n'Play is capable of directly monitoring the attested device instead of comparing hash values of known benign states. This has multiple advantages: A preliminary investigation of all valid states is not needed, and more complex checks are possible, for example, bounds checks. Furthermore, the DMA'n'Play framework is also suitable for existing legacy devices as neither specialized hardware components nor the source code of the attested application is required. In contrast to software-based attestation, the implementation of DMA'n'Play is straightforward. In fact, the changes to the software of the attested device to integrate DMA'n'Play are minimal as the attested device only needs to send its raw memory contents.

We implemented DMA'n'Play in two real-world examples, a medical device, and a drone, and showed in full end-to-end examples how DMA'n'Play can be used to detect compromises of configurations. Furthermore, we provide integration guidelines that explain how DMA'n'Play can be integrated into new or existing devices and how to develop configuration files for the verifier to define benign states. In addition, we propose DMA'n'Play To-Go, a tiny low-cost embedded device that can be used instead of the verifier. It can either take over the role of the verifier but also allows to relay the data from the DMA bus to an external verifier. This allows to attest mobile devices during operation, such as vehicles or drones.

CHAPTER 6

Hotpatching of Real-Time Applications

By design, remote attestation has a significant limitation. It solely allows the detection of attacks or compromises of devices. When remote attestation detects a compromise, an adequate reaction is required, for example by restoring a benign state, stopping the affected device, or removing compromised devices from a swarm or network of devices. However, remote attestation is not able to prevent the actual attack of a device. To prevent an attack its root cause needs to be resolved. This is typically done by means of a patch or update that corrects the underlying error, thereby eliminating the root cause of the attack.

Updating software is a standard feature of computer systems and applications and has become a standard task in IT. In fact, today, patch management and distribution of patches belong to the main and common tasks of IT departments [120]. Patching is also an important measure to increase the security of IoT deployments [12]. The widely recognized MITRE ATT&CK[1] framework [281] describes regular software updates as mitigation for industrial control systems (ICS), addressing a wide range of attacks, including the severe drive-by attacks, privilege escalation, and the exploitation of remote services [272]. Patching is not only important for commodity computer systems such as servers and desktop computers, and embedded devices such as IoT devices, but also for highly safety-critical systems like medical devices. Recall that embedded devices often suffer from manifold vulnerabilities, as eluded in Chapter 1, including industrial robots [218], vehicles [161], and drones [13].

A recent study showed that even medical devices, in particular pacemakers are vulnerable [133]. Pacemakers are medical devices permanently implanted in patients' bodies to regulate the heartbeat when the heart is beating too slowly or irregularly. The devices send electrical signals to the heart to stimulate it to beat at a regular rhythm. Modern pacemakers are programmable and can be adjusted to the patient's needs and monitor parameters of the heart. While these devices are life-sustaining, a malfunction can be life-threatening. Patching these devices is possible but challenging as it can interrupt the functionality or cause unwanted side effects.

[1] `https://attack.mitre.org/`

For instance, in 2017 the FDA (Food and Drug Administration) approved a firmware update for a pacemaker to fix severe security vulnerabilities that affected more than 450,000 patients [275]. The exploitation of these vulnerabilities allowed an attacker to obtain unauthorized access and issue commands or modify the settings of the pacemaker [78]. The update itself was non-invasive. However, it came with side effects. During the reprogramming of the device, there was a fallback to ventricular demand pacing. For some patients, this may cause temporary symptoms. Furthermore, the vendor stated that the update will cause some unpredictable failures and device resets. These unpredictable events could have severe consequences, and even be fatal [162]. The manufacturer advised, that patients dependent on pacemaker should consider having this upgrade performed in a location where temporary pacing and pacemaker generator can be readily provided [275].

The pacemaker is only an example of a whole class of devices that fulfill safety-critical tasks that cannot be interrupted, often because they interact with the physical world. As shown in Chapter 2, there are many areas in which such devices perform highly critical tasks, and where device failure has severe consequences, such as in vehicles, industrial machines, robotics, and medical devices.

While the installation of updates is a standard procedure, this process usually requires a reboot or restart of patched services, thereby interrupting the service of the patched device or application. In general, the rate and speed of patch deployment are significantly affected by the update method used [191]. Patches for IoT devices are usually relatively large as they typically contain the device's full firmware, and need to be applied in a single step without interruptions due to the architecture of the devices. This typically results in system downtime due to the need for an interruption of the normal operation to apply the update and a loss of state due to a reboot. As a result, updates must be planned. For instance, it is necessary to wait for acceptable downtime or patching windows, thereby delaying the patch application. Sometimes, devices even do not accept any downtime as they need to operate continuously, for example in control units in plants [95] or medical devices such as the aforementioned pacemakers [162].

However, this creates a window of opportunity for attackers to exploit unpatched devices, even when patches have already been released. Although this may initially seem like a minor issue due to the limited time window, recent studies have shown that exploits for a given vulnerability are often publicly available within a day of the patch being released [243]. So, this presents a serious security concern for IoT systems, especially considering devices with safety-critical tasks, where a compromise can have serious consequences.

A solution to this problem is hotpatching. Hotpatching is the application of patches during runtime. This way, the system remains fully available at all times during the patching process, as the normal operation of the system is not interrupted, no processes are stopped, and the system state is preserved. Research focused on hotpatching of traditional service [238] and in particular server applications [63, 122, 213]. However, hotpatching of embedded devices and, in particular, systems with real-time constraints, have not been given much consideration. Hotpatching is a complex process as it is performed while the system remains running in parallel. Thus, it must be ensured that there are no interactions between the patching process and the normal operation, so that the patched system remains functional at all times.

Hotpatching of systems with real-time constraints faces additional challenges. While in hotpatching of commodity computer systems, the timing behavior is less relevant [238], for real-time systems the timing behavior is crucial. As hotpatching influences the operation of the patched system, typically by adding code fragments, it must be ensured that the timing behavior of the patched system remains within all thresholds. During the development of real-time systems, the timing behavior is verified in the so-called profiling phase. So, a hotpatching solution for real-time systems must have a small and deterministic overhead. Large delays or unpredictable behavior of the patching process can result in missed deadlines of the real-time system and hinder profiling during patch development. Despite all these challenges, hotpatching remains the only solution to apply patches to resolve security vulnerabilities during runtime, when downtime is not an option.

Contributions. In this chapter, we present a new hotpatching framework called HERA *(Hotpatching for Embedded Real-time Applications)*. HERA leverages a common debug feature of commercial off-the-shelf processors to activate newly applied updates in real-time with predictable and minimal overhead. Using hardware breakpoints, we can activate patches in a single processor instruction, without the need to modify the installed firmware. This way, the timing behavior of the patching process remains precisely predictable, so that no real-time constraints are violated. Unlike existing methods such as Katana [219] and Kitsune [136], our approach does not require dynamic linking or prior modifications to the target program. Furthermore, we do not need to modify the installed firmware. Firmware modifications during runtime are challenging to the monolithic architecture of the firmware, and the block-writable flash memory of embedded devices.

In the HERA hotpatching process, the patch is prepared in the background during idle time, without interfering with real-time critical tasks. Once the patch has been applied, it is activated using a single processor instruction using the breakpoint unit that is implemented in hardware. The overhead to switch to the patched section is minimal, as this is done using a hardware-based processor feature and a trampoline to jump to the corresponding code section.

There are two methods to develop hotpatches for HERA. The first method is to develop the patches in higher programming languages and then derive the hotpatch from the compiled binary using binary diffing. This approach is preferred when the source code is available. The second method involves creating patches directly on the assembly level. This allows the development of patches for closed-source devices when no source code is available.

We show how to implement the HERA framework and give detailed integration guidelines. In a comprehensive evaluation, we implement HERA in two critical embedded devices, a syringe pump and a heart rate sensor to show its applicability. In addition, to show its practicability, we use HERA to patch a real-world vulnerability in the popular FreeRTOS real-time operating system. Furthermore, we perform extensive measurements to verify the runtime properties of the HERA hotpatching strategy. Using an oscilloscope, we measure the runtime overhead during patch application and show that HERA is even applicable to systems with hard real-time constraints. This is the class of the most critical real-time systems.

In summary, we provide the following contributions.

- We propose HERA, the first hotpatching framework that uses internal hardware debugging features to apply patches instantaneously, which is suitable for real-time devices, even with hard real-time constraints. HERA considers the architecture of embedded devices, such as block-writable memory, and a monolithic, static software architecture. HERA does neither require hardware changes nor source code of the patched system.

- In addition, we line out detailed implementation guidelines that explain how to integrate the HERA framework into existing applications and how to adapt existing patches to create suitable hotpatches.

- We show the applicability of HERA. Using the HERA framework, we patched two real-world medical devices with real-time constraints, namely a syringe pump and a heartbeat sensor. Furthermore, we used HERA to patch a vulnerability in the FreeRTOS operating system.

- In an extensive measurement study, we verified that HERA is suitable for systems with hard real-time constraints due to its predictable behavior as well as its deterministic, minimal overhead when applying the patch.

6.1 Background: Hotpatching Strategies

Hotpatching means applying a patch during runtime, without disrupting the normal operation. One main challenge in hotpatching is activating the patch while maintaining the program state. In any scheme, eventually, the execution of the application needs to be altered during runtime to activate the patch. There exist three different approaches to implementing hotpatching on embedded systems.

6.1.1 Relocatable Executables

One approach to hotpatching is to modify the links between program components. Dynamically linked applications reference code that is not included in the compiled library. During runtime, these links are dereferenced and then loaded. This is in contrast to statically linked libraries, which are included during compile-time and therefore are a fixed part of the compiled binary. Dynamic linking is a common concept for shared libraries in modern operating systems such as Windows and Linux. The operating system holds a data structure containing symbolic links to the shared code and resolves those links during runtime. These dynamic links can be used for hotpatching: To replace application components, only the references have to be changed. This allows the exchange of whole components while maintaining an unlimited number of total active patches. This is a popular approach to implementing hotpatching. For example, the frameworks Katana [219] and Kitsune [136] require dynamic linking. While dynamic linking is common for commodity computer systems, embedded devices are often statically linked. Especially embedded systems with real-time constraints are typically linked statically due to the need for predictable runtime and reduced overhead. Dynamic linking can create overhead and may reduce predictability, a key property of a real-time system. For example, the FreeRTOS real-time operating system is statically linked [137]. Consequently, such systems cannot use patching frameworks that necessitate dynamic linking.

6.1.2 Instrumentation

Another approach to implement hotpatching is directly modifying the program during runtime. In the case of embedded devices, this implies changing the device's firmware. This is often implemented using so-called trampolines [303]. Trampolines are simple instructions such as branch or jump that allow the insertion of complex changes without many modifications to the original application. However, on embedded devices, this approach is limited by architectural properties. Embedded devices often feature block-writable memory, so whole blocks must be written at once, resulting in large overhead, see Section 3.7. While these constraints on the one hand allow implementing the attestation schemes in Chapter 3 and 4, on the other hand this is a limitation when performing patching during runtime. Especially in the case of systems with guaranteed responsiveness, the delays incurred by writing whole blocks in the flash memory make the patching process inadequate for systems with real-time constraints. While in this approach instructions are inserted into the application before being executed, this can alternatively also be done during runtime.

In dynamic binary translation, the application is executed in a translator component that is capable of replacing or adding instructions on-the-fly [212]. The hardware architecture of embedded devices does not limit this approach, as it does not need to modify the firmware. However, dynamic binary translation generates overhead and requires additional system resources, which are limited to embedded devices. Furthermore, the profiling must be repeated when implementing dynamic binary translation into legacy systems with real-time constraints due to the added overhead. This is to ensure that all timing deadlines are met at all times, even though the original binary has not been altered.

6.1.3 A/B Hotpatching

A/B patching requires maintaining two instances of the patched system: one instance that is currently active and running, and another instance that is available for updates. The general idea is that the updates can be performed on the inactive instance, independent of the running instance. When the update process is finished, a switchover is made from the active instance to the newly updated instance [168, 190]. The main advantage of this approach is that it reduces downtime, as the switch can be made quickly and seamlessly without interrupting the system's operation. The duration of the patching process itself does not influence the normal operation, as it is done on the inactive instance. However, the A/B patching method incurs significant overhead: It requires twice the amount of memory to hold both instances, as well as a dedicated management mechanism to perform the update and the switchover. A/B patching is commonly used in practice for over-the-air updates on embedded and mobile devices. For example, the ESP32 uses an A/B updating scheme [100], and system updates of modern Android smartphones also use an A/B method [20]. However, both systems require a full reboot to perform the switchover to the patched instance and thereby activate the update, thereby interrupting the normal operation. When using A/B patching for hotpatching, the state of the active and the patched system need to be synchronized to allow a seamless switchover, making the implementation of the switchover a complex task compared to the reboot-based approach. Furthermore, due to cost reasons, embedded devices usually have minimal hardware and memory capabilities, making maintaining two copies of the system infeasible. The A/B hotpatching approach can be extended further by using multiple instances of the same system in parallel. This way, it is possible to take a single instance offline, apply the patch, and take it back online, while the other instances continue to operate normally [62]. But, this method requires even more resources as these instances have to be executed in parallel.

6.2 Challenges

Hotpatching is patching a system during runtime. This is a complex process that requires careful implementation to avoid interference with normal operation. It is crucial to ensure that the changes are applied correctly and do not cause any unintended behavior or system instability. Especially in the realm of systems with safety-critical constraints, the patching process may not impair the correct functionality of the device in any way. In systems with real-time constraints, all timing constraints have to be met before, while, and after applying the patch. Especially in systems with hard real-time constraints, there may not be any circumstances in which the deadlines are missed. Thus, the behavior of the patching process must be deterministic, to guarantee compliance to functional and timing constraints. Integrating hotpatching into legacy devices poses further challenges, as these devices' software and hardware architecture is predefined and cannot be changed easily.

Despite these challenges, hotpatching is the only solution to react to newly discovered critical vulnerabilities when regular patches cannot be applied due to the need for downtimes or system reboots. Because hotpatches are applied during runtime, it is important to consider the state of the patched system. This is in contrast to traditional patching, where the state is reset to a determined condition with a restart of the module, application, or whole system.

In particular, hotpatching embedded systems with real-time constraints poses the following challenges.

Challenge 1: Continuous Operation. The patched system must continue to operate normally. Neither the application of the patch, that is, the downloading and integration of it into the system, nor the activation of the patch may interrupt or impair the operation of the system.

Challenge 2: Predictable Timing and Real-Time Constraints. The correct operation of a real-time system also includes its timing behavior. So, a hotpatching scheme for real-time systems must obey all timing deadlines of the patched system. This means that tasks related to applying the patch, but also activating and running the patched components, may not negatively impair the timing behavior of the patched system. Even under worst-case conditions, timing deadlines may not be missed.

Challenge 3: Deterministic Behavior. Systems with safety-critical tasks and hard real-time requirements must fulfill these constraints under all circumstances. This is verified during development in the so-called profiling phase. Thus, these devices need a deterministic behavior to ensure these constraints. This is especially challenging in complex systems with many parallel tasks, all possible states must be considered to prevent concurrency issues.

Challenge 4: State Preservation. Hotpatches are applied on a running system. During runtime, the state of the system, which includes the memory content and the currently executed commands, constantly changes. When hotpatches are developed, this state must be preserved and may not be corrupted to ensure constant availability and no service interruption. Moreover, when a patch requires state changes, the hotpatching scheme needs to perform these without interrupting the system's normal operation.

Challenge 5: Legacy Devices. Developing a patching method for legacy devices induces challenges on both the software and hardware level. The patching scheme needs compatibility with existing hardware, as hardware changes in legacy devices are often not an option as we showed in Chapter 2. In addition, the patching process must work with few hardware requirements. Embedded devices are used in large numbers; therefore, they are optimized to use minimal hardware. They are also designed to consume little energy, because embedded devices often operate in environments with a limited energy supply, for instance, when they are battery-powered. As a result, existing embedded devices often have little spare hardware resources and memory. Thus, approaches such as A/B patching are not applicable on this type of device. In addition, during runtime, changes in the software architecture of legacy devices are often infeasible. Embedded devices typically feature a single monolithic firmware block. Furthermore, this firmware is statically linked to ensure a deterministic behavior due to real-time constraints. Often, source code of these applications is even not available. In these cases, patching based on binary firmware is the only option.

Challenge 6: Patch Development. Ideally, a hotpatching scheme allows applying existing normal patches as hotpatches, thereby reducing the effort for developing, implementing, and testing the patches. Especially in safety-critical systems where the correct functionality is an important aspect, testing and verification require a significant amount of work. Hotpatching schemes that require additional development are less likely to be adopted.

To the best of our knowledge, there does not exist a hotpatching solution for embedded devices that addresses all of these challenges. Prior research focuses on the correctness and safety properties of the patch application, reducing downtime and resource usage, or requiring dynamic linking which embedded devices often do not feature. For instance, the Cetratus framework [190] uses a quarantine mode to initialize, test, and monitor a live patch. The hotpatching frameworks Katana [219] and Kitsune [136] both require dynamic linking and thus cannot be used on real-time embedded devices that are statically linked. We review further hotpatching approaches in the related work in Section 6.9.

Our solution HERA addresses all of these challenges. HERA uses widely available hardware debugging features of typical embedded system processors to apply hotpatches during runtime to replace instructions on-the-fly. Furthermore, in contrast to other hotpatching schemes, such as A/B patching, the memory overhead is minimal as the active instance is patched directly. Therefore, HERA is also suited for legacy devices. Finally, the usage of these hardware features that allow predictable instruction replacement minimizes overhead for the patch application while maintaining deterministic behavior, a crucial requirement for systems with hard real-time constraints.

6.3 Assumptions and Threat Model

In the following, we describe the prerequisites to implement HERA hotpatching on embedded devices.

6.3.1 Assumptions

We assume an embedded device with real-time constraints. The device features a real-time operating system (RTOS) architecture that accommodates different types of deadlines, including hard, firm, and soft deadlines. The RTOS correctly schedules all tasks and ensures that all deadlines are met. The system has a vulnerability that needs to be patched. We assume that a patch is available and that the system has sufficient resources to run the patched system. So, the patched system must still be able to function correctly, including meeting all deadlines of time-critical tasks. In addition, the embedded device features a hardware-based method to perform instruction replacement, as described in Section 6.4, to implement the atomic switchover, such as the Flash Patch and Breakpoint (FPB) unit of ARM M3/M4 processors.

We assume a secure updater that is capable of performing the update process. The system has idle time or low-priority tasks that can be rescheduled to allow the download and processing of the patch in the background, as well as the memory required to do so. Therefore, a secure updater module is able to check for an update, as well as download, verify its integrity, and apply the patch in the background without violating real-time constraints. Furthermore, the updater can trigger the switchover of the HERA framework. As the updater needs to be adapted to its use case, the updater itself is not part of the HERA framework. Such updaters are a common feature of IoT devices [12]. For instance, FreeRTOS already features an updater that allows over-the-air (OTA) updates [114] and includes functionality to (1) receive, parse, and validate the update request, (2) download and verify the file according to the information in the update request, (3) run a self-test before activating the received update to ensure the functional validity of the update, and (4) update the status of the device. This updater service can be adapted to the requirements of HERA.

6.3.2 Threat Model

We assume that the device has a security vulnerability, caused, for example, by a programming flaw. A remote attacker is able to exploit this vulnerability. Furthermore, we assume that a patch is available to fix the vulnerability and mitigate the root cause. The patch may be available as a binary file or as source code. Typically, such patches are simple and featureless [14]. We assume that this patch can be applied by adding or replacing code blocks, for instance, to change statements, add checks, or validate inputs. We exclude complex feature updates, that change large parts of the firmware. We assume the update process to be secure. The attacker is unable to compromise the updater service. So, we assume that the updater itself does not have any vulnerabilities, and all updates are checked for integrity and authenticity before being applied. We exclude physical attacks on the device.

6.4 Concept of HERA

The main concept of HERA is to use common hardware debugging features of embedded devices to implement hotpatching on these systems. We use these debugging features to perform the switchover to patches with minimal overhead while maintaining deterministic behavior. In particular, we use an instruction replacement functionality to perform dynamic trampoline insertion in the execution of the program binary. This way, we can jump to arbitrary code sections within a single processor instruction. Thus, we do neither need to modify the program binary to apply patches nor introduce overhead due to dynamic translation or additional memory for A/B patching schemes. This design makes HERA suitable for embedded devices with timing requirements, including the most critical systems with hard real-time constraints.

The instruction replacement feature that HERA uses is widely available in commercial of-the-shelf processors, such as the ARM Cortex-M3/M4 and the ESP8266, and its successor ESP32. Hence, HERA can also be implemented on legacy devices, as HERA only requires changes to the software. HERA works on standard real-time operating systems (RTOS) and can be integrated into the updater services of commodity embedded devices. This makes HERA a suitable solution for legacy devices, as neither their hardware nor their software architecture needs to be changed.

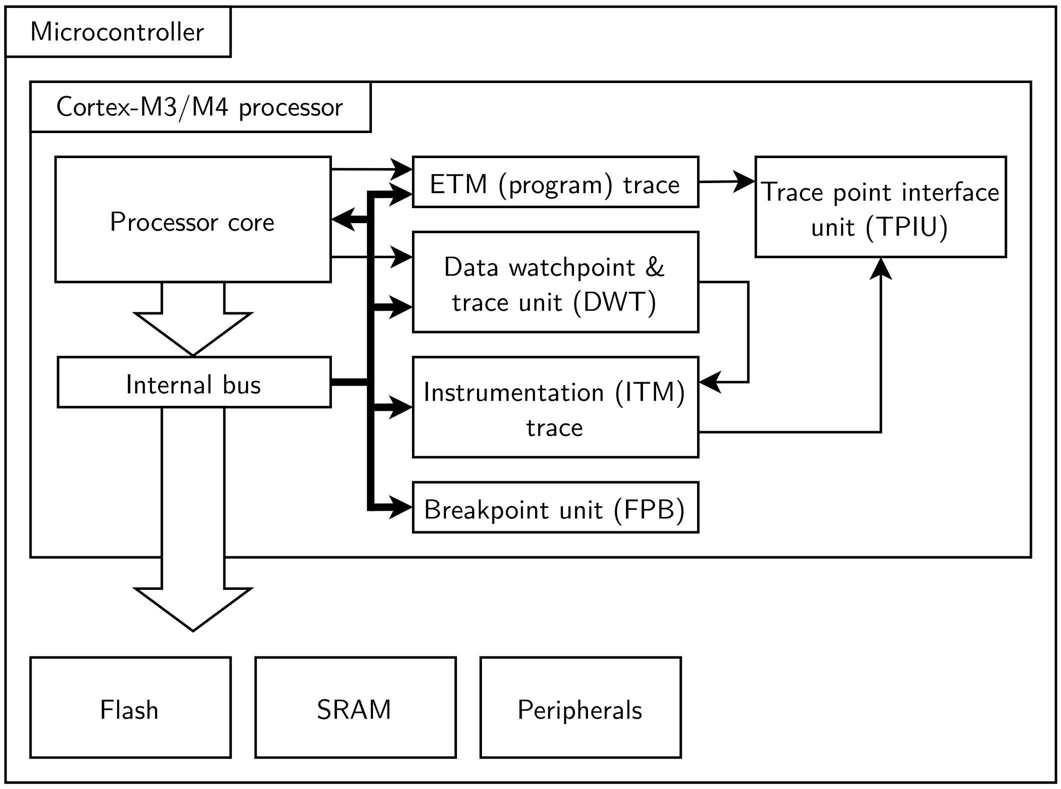


Figure 6.1: The FPB unit in ARM Cortex-M3/M4 processors is connected to the CPU via a bus [293].

In the following, we discuss the functionality of the hardware debugging unit. Then, we explain the design of HERA. After that, we describe the different components of the HERA framework and the complete patching process.

6.4.1 Hardware Debugging Units

Commodity microcontrollers in embedded devices have several debugging features. One widespread feature allows inserting breakpoints for debugging purposes. Developers can set breakpoints to halt and resume the execution of the processor. This way, developers can examine the state of the processor, memory, and peripherals at specific moments during execution. This method is typically used during development to identify bugs. Hardware debugging units can also have other features. Another widespread feature is an instruction replacement functionality that allows the replacement of a single instruction.

For example, the ARM Cortex-M3 and M4 processors feature a so-called flash patch and debugging (FPB) unit, which has such an instruction replacement functionality [25]. Figure 6.1 shows the architecture of such processors [293]. The FPB unit is connected with the processor core via a bus. The FPB unit can interrupt the normal execution of the processor, drop the fetched instruction, and replace it with a different instruction. During execution, a comparator checks for breakpoints. When a breakpoint is reached, the execution is halted or instructions are replaced. This functionality is implemented in hardware. A full FPB unit can handle up to six breakpoints [292]. The Cortex-M3/M4 processors are good examples of typical processors in embedded devices. ARM is the most popular processor architecture. More than 100 billion processors with an ARM architecture have been shipped [30]. The ARM processor architecture is especially popular for embedded and IoT devices [30, 88]. Many devices with real-time constraints are built using such processors. For example, many programmable logic controllers (PLC) that are used for control tasks in industrial applications use ARM processors [237].

In HERA, we use the instruction replacement functionality integrated in commodity processors to apply patches during runtime. We redirect the control flow to the patches without impairing the systems' performance. As the instruction replacement is performed in hardware, the overhead is minimal and constant. This functionality is not limited to the FPB unit of ARM Cortex processors but is a widespread feature of hardware debugging units. For instance, the Tensilica Xtensa processor architecture has a similar debug functionality [269]. Tensilica Xtensa processors are used in Espressif ESP8266 and its successor ESP32 microcontrollers, which are a popular basis for IoT devices due to their integrated Wi-Fi functionality [99, 103]. Espressif shipped more than 100 million of such chips [96].

6.4.2 Patching Process

The HERA hotpatching framework uses the onboard debugging unit of embedded processors to apply hotpatches. Using the instruction replacement functionality, HERA is able to modify the binary during runtime without actually modifying the binary. HERA uses breakpoints to insert trampolines into the application during runtime. With these trampolines, the execution is redirected to patches. As the jumps to the patches are done in hardware, these instructions are atomically and cannot be interrupted. The number of different trampolines is dependent on the number of breakpoints available in the debugging unit.

Figure 6.2 shows the general architecture of HERA. The HERA patching process consists of two phases. In the first step, the patch is prepared. The updater process fetches the update, verifies its integrity, unpacks and copies it to the correct memory sections. This is done during idle time when no tasks are scheduled, or by rescheduling low-priority tasks. In the second step, after successfully preparing the patch, the

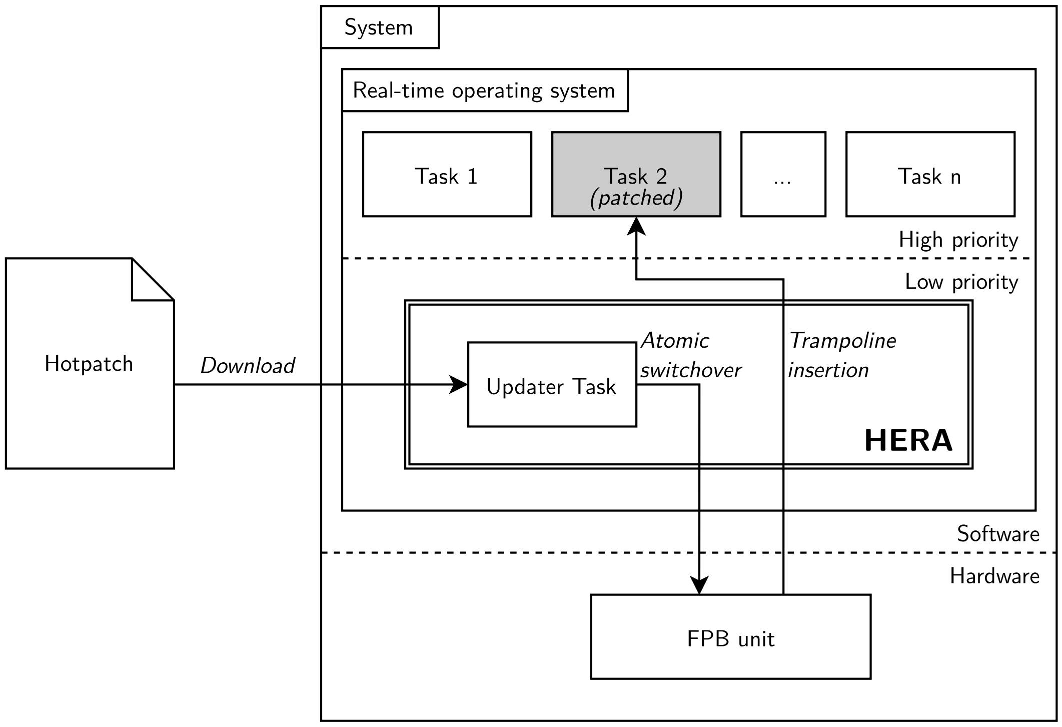


Figure 6.2: Concept of HERA hotpatching. After the updater has downloaded and prepared the patch, the hardware-based flash patch and breakpoint unit (FPB) triggers a switchover to the patched region using a trampoline.

updater activates the patch. This is done in an atomic switchover using the hardware breakpoint unit. Once the breakpoint unit has been configured, the hotpatch is active. The processor inserts a jump to the patch section when the breakpoint is reached using the corresponding trampoline function. HERA works on standard real-time operating systems (RTOS) that schedule all tasks according to their priorities and deadlines. The RTOS also schedules the HERA updater process that fetches and prepares the patch to run in the background without impairing the real-time tasks. Note that this patch preparation step is not time-critical and hence can be interrupted at any time as it does not influence the normal operation.

Figure 6.3 shows an example of the execution of the patching process. The system runs tasks of different priorities. When no tasks are running, the processor is in idle state. During this time, the patch can be prepared. This process is interrupted multiple times to execute other tasks with low and high priorities. The RTOS performs the scheduling of these tasks. After the patch preparation has been completed, the hotpatch is activated. When the breakpoint is reached, a jump to

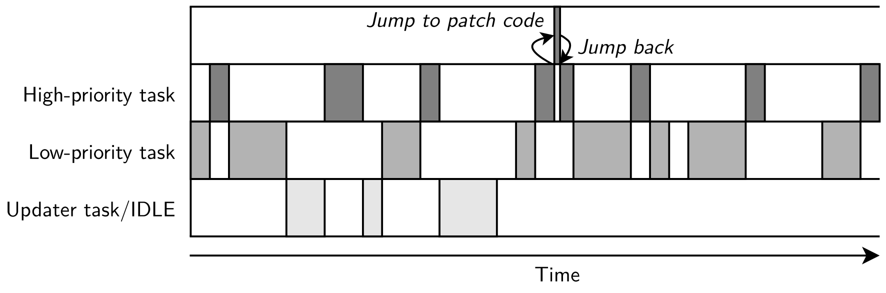


Figure 6.3: Sequence view of the patching process. The updater runs in the background during idle time. When the patch is activated, there is an atomic jump to the patched code.

the patched code is inserted using a trampoline. This process seems like a standard patching process with trampoline injection [303], as described in the background in Section 6.1.2. In HERA however, this insertion is done on the hardware level, inducing minimal and predictable overhead, in contrast to software-based approaches.

In the following, we describe the two steps of the patching process in detail.

Patch Preparation. The updater task performs the patch preparation step. The updater task fetches the update, typically by checking for new updates in regular intervals and then downloading the updates as soon as they are available. The patch consists of two parts: First, the actual hotpatch, which consists of pre-compiled code for the target architecture. Second, the meta-information on the hotpatch, which contains all information necessary to apply the code of the patch, such as insertion points and code sizes. Typically, both parts are packed together in a container format and signed to allow the verification of the integrity and authenticity of the patch. The updater unpacks the patch and copies the patch code to a patch slot in RAM. Then, the updater configures the breakpoint unit using the meta-information of the patch and prepares the trampoline code for the jump to the patch. On some system architectures, it is not possible to redirect the control flow directly to the RAM, where the patch is stored. For such cases, HERA features predefined trampolines in ROM. A dispatcher selects the correct trampoline using the control flow information on the stack. In summary, the patch preparation process covers the following steps:

1. Checking for updates and download of the patch.
2. Verification of the authenticity and integrity of the patch.
3. Extraction of the meta-information from the patch.

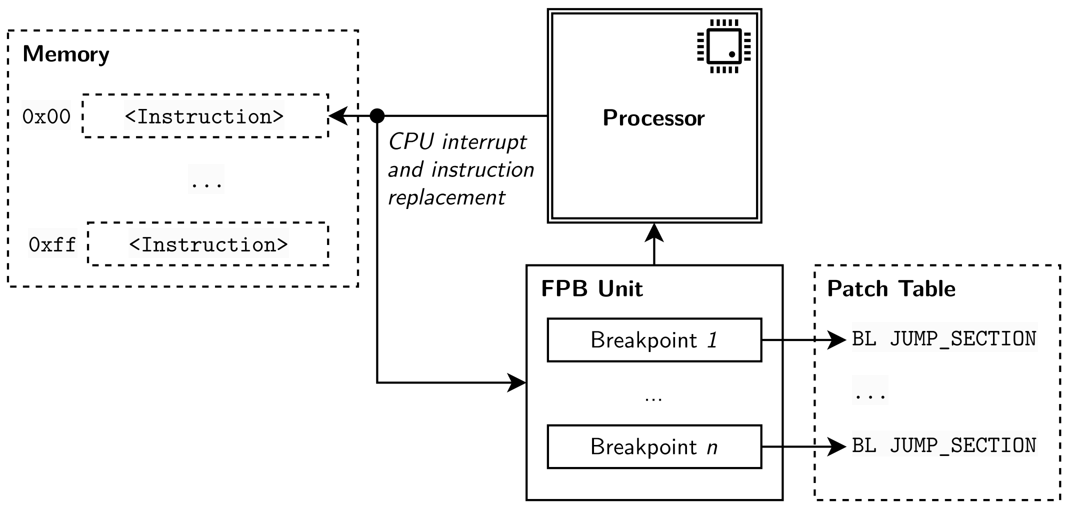


Figure 6.4: The processor is able to insert trampolines using the instruction replacement function of the FPB unit.

4. Copying the patch into a patch slot in the memory and performing the patch preparation.
5. Setting up the trampoline for the patch. Alternatively, if direct jumps from ROM to RAM are not possible, the addition of an entry to the dispatcher.
6. Configuration of the breakpoint unit with the insertion point and branching instruction.
7. Activation of the patch by triggering the atomic switchover.

In the following, we explain how the patch is activated using the atomic switchover and how these breakpoint units work in detail.

Patch Activation Using the FPB. When the patch preparation is completed, the patch can be activated using an atomic switchover. This switchover is atomic, as it only requires writing one register, which is done with a single assembly instruction. By definition, a single assembly instruction is atomic as it cannot be split in software. To insert a patch, a breakpoint needs to be configured. Breakpoints are typically used for debugging purposes. These breakpoints can be configured using specific hardware registers. HERA uses these breakpoints to insert patches during runtime. Figure 6.4 shows the functionality of the breakpoint unit. The breakpoint unit monitors all the instructions that the processor executes. For each breakpoint, the breakpoint unit has a comparator. In this way, the breakpoint unit can perform

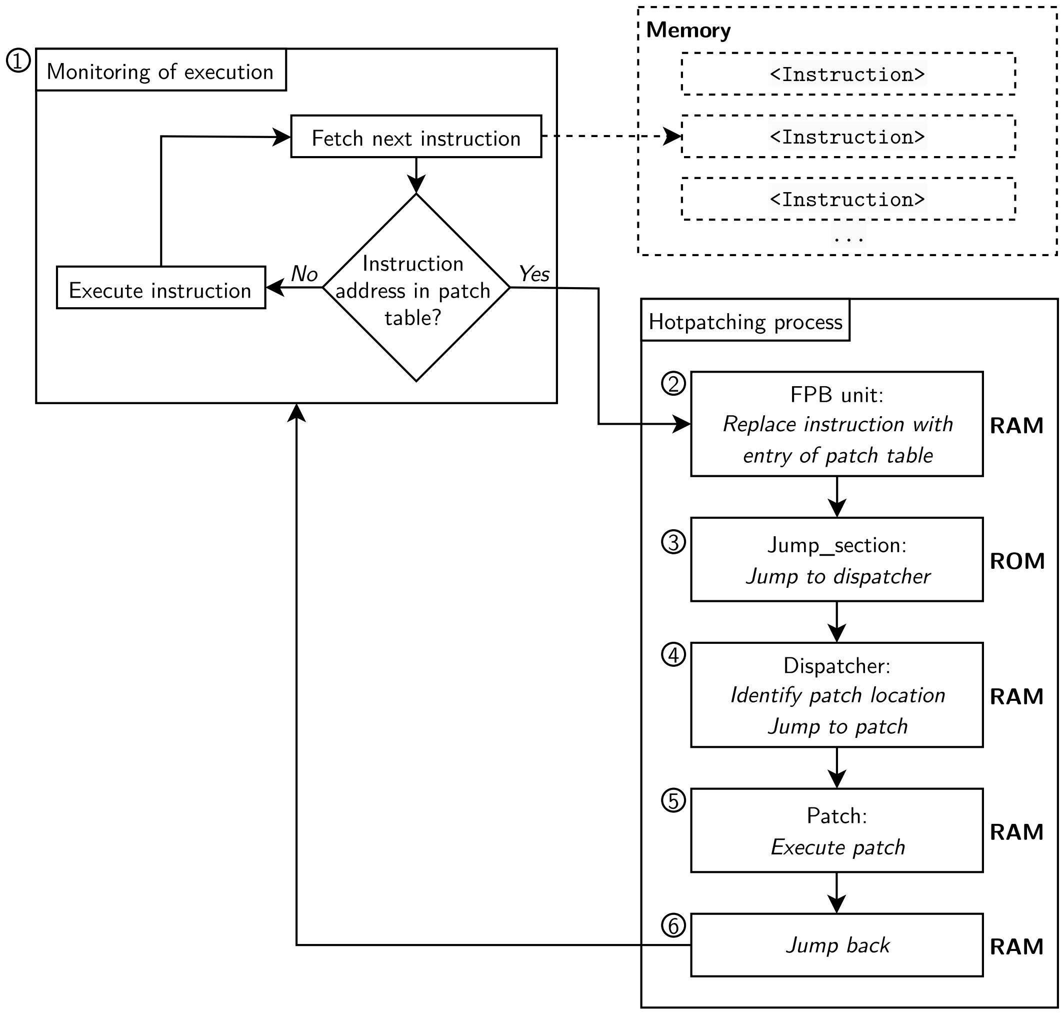

Memory
<Instruction>
<Instruction>
<Instruction>
...
① Monitoring of execution
Fetch next instruction
Instruction address in patch table?
No
Yes
Execute instruction
Hotpatching process
② FPB unit: Replace instruction with entry of patch table
RAM
③ Jump_section: Jump to dispatcher
ROM
④ Dispatcher: Identify patch location Jump to patch
RAM
⑤ Patch: Execute patch
RAM
⑥ Jump back
RAM

Figure 6.5: The process flow of the trampoline insertion using the dispatcher.

hardware-based identification of breakpoints. Breakpoints are identified by memory addresses. When a breakpoint is reached, the breakpoint unit can interrupt the execution, for example, to enable debugging. In HERA, we use this breakpoint to insert trampoline instructions to jump to a patch. These replacement instructions are stored in the patch table. Each breakpoint references one instruction that is replaced. During this process, the system continues to operate normally; the execution is not delayed as the debugging unit is implemented in hardware. Furthermore, because these interrupts are handled in hardware, this hotpatching process is transparent to the patched system. Breakpoints are independent of each other, thus it is possible to reconfigure breakpoints to add or modify hotpatches without impairing other hotpatches.

The process of replacing instructions and applying patches is depicted in detail in Figure 6.5, focusing on the switches between RAM and ROM. These switches are crucial for patch application, as modifying the firmware of embedded devices during runtime is challenging, as eluded earlier. Often, the firmware is only block-writable, or even not writable at all during runtime. Therefore, HERA uses RAM to store hotpatches, which is fast to write with fine granularity. The programming models for debugging units differ between processors, although they have the same instruction replacement functionality. Therefore, in the following we focus on the flash patch and breakpoint (FPB) unit of ARM processors, which we also use for our implementation of HERA.

1. The breakpoint unit continuously monitors the instruction that the processor executes by comparing the respective memory addresses with the configured breakpoints

2. When a breakpoint is triggered, the current instruction is aborted and replaced with the corresponding entry from the patch table.

3. The dispatcher is called using a branch-and-link instruction.

4. The ARM Cortex processor does not support direct branching from ROM, where the original application is stored, to the hotpatch in RAM. Because of these range limitations, we use a dispatcher in RAM that allows us to perform this switch indirectly.

5. Then, the patch is actually executed.

6. After the execution of the patch, the control flow finally returns to the original location and continues the normal execution of the application.

6.4.3 Limitations

The goal of HERA is patching critical security vulnerabilities during runtime without interrupting the normal operation, and while maintaining all timing constraints. These vulnerabilities are for example caused by common memory errors. To mitigate them, typically additional checks need to be introduced, or code segments must be replaced. Nevertheless, HERA can also be used to add additional functionality, for instance by adding new code parts. However, hotpatches should remain simple, as complex changes need a verification that the state of the application is not corrupted. During runtime, the internal state, that is, the program state and memory contents, are constantly changing. So, modifying this state is a sensitive task, which should be kept as simple as possible. In practice, this is no limitation, as security patches are typically small and do not contain new features [14]. But to support complex changes to the memory layout the HERA framework needs to be extended to ensure consistent state changes.

By design, HERA hotpatches are not persistent. This means, that these patches are removed upon reboot. This is an intentional design, as writing to non-volatile memory on embedded devices is often not feasible and, in addition, typically incurs significant overhead due to the usage of block-writable flash memory. In order to keep HERA hotpatches during reboots, the updater service needs to be adapted such that it loads and applies the hotpatches upon start of the device. However, if writable permanent memory is available, HERA can also be modified to use this memory to implement permanent hotpatches.

Another limitation of HERA is the number of active patches. For each patch, the processor needs to maintain a separate breakpoint. The number of active breakpoints a processor can keep is limited by its hardware. For instance, the flash patch and breakpoint unit of the ARM Cortex-M3/M4 processor features six slots for breakpoints [292]. However, not every patch needs a dedicated breakpoint. Combining multiple patches into one breakpoint or addressing multiple vulnerabilities in one patch is possible. The breakpoint is only the entry point to the patch and does not limit the size of the patch. In addition, HERA focuses on hotpatching. As HERA hotpatches are non-permanent, the hotpatches should be replaced by permanent, traditional patches at the earliest convenience, freeing up breakpoints for new hotpatches. In practice, on typical embedded devices a full firmware update will be applied during the next patching window or planned reboot of the system. After this, all breakpoints are available again.

6.5 Implementation

To show the applicability of HERA, we performed an extensive case study. We integrated HERA into two real-world medical devices that have strict real-time constraints and used HERA to patch a critical real-world vulnerability in the popular FreeRTOS real-time operating system. We implemented HERA using the popular ARM Cortex-M4 processor that features a flash patch and breakpoint unit. In the following, we focus on the implementation of HERA on such processors. The programming model of other debug units may differ, so porting HERA to other platforms may require adaptations. However, only the hardware-specific parts of the framework need to be changed. For the implementation, we used a STMicroelectronics NUCLEO-F446RE development board, that features a Cortex-M4 processor, 128 kbyte RAM, and 512 kbyte of flash memory. This is a typical hardware configuration of embedded devices. First, we describe the main functions of the HERA hotpatching framework. Second, we explain how to develop compatible hotpatches. Then, we show how this hotpatch is applied. To illustrate the functionality of HERA, we show how we developed a patch for a critical vulnerability in the FreeRTOS real-time operating system as an example. In the evaluation in Section 6.7 we give more details on the integration in the medical devices.

6.5.1 HERA Library

The main component of HERA is the HERA library. This library is implemented in C and supports the implementation of hotpatching on embedded devices using common hardware debugging features as eluded earlier. To integrate a hotpatch, besides the actual patch, three things are required: A hardware breakpoint to perform the trampoline insertion, the trampoline to deviate the control flow to the hotpatch, and the insertion point, that is the location in the original application where the patch is integrated. The HERA library provides functions to configure the flash patch and breakpoint (FPB) unit, loads patches, integrates a dispatcher, and activates the patch by performing an atomic switchover. In particular, the HERA library has the following main functions:

`fpb_init`. First, the fpb_init function is called. It checks whether a compatible FPB unit is available. If so, it initializes the FPB unit by preparing and referencing the necessary data structures and then confirms a successful initialization.

`fpb_enable`. This function enables the FPB unit by setting the corresponding configuration bit. It can be used to enable or disable HERA globally.

`enable_single_patch`. This function can enable a patch. First, the function creates a trampoline to a hotpatch. The trampoline consists of a branch instruction. There are two options to specify the target of the branch instruction. Either, a patch can be supplied. Alternatively, the target can be calculated based on given offsets. Furthermore, the function configures the breakpoint to insert the hotpatch. After completing these preparation steps, the function performs an atomic switchover to activate the hotpatch. This is done by setting the register of the breakpoint.

`load_patch_and_dispatcher`. This function loads and prepares the patch. First, the function loads a simple dispatcher to allow indirect jumps to the hotpatch in RAM. This dispatcher is then configured according to the meta-information provided with the patch. Finally, the patch is copied to the correct location in RAM.

The resulting patch application process has the following steps.

1. The availability of a suitable FPB unit is checked.
2. The FPB unit is initialized
3. The dispatcher is loaded into the RAM
4. The patch itself is also loaded into the RAM.
5. Now, the optional dispatcher can be configured accordingly. The dispatcher is only needed if required due to architectural limitations, as in our case study.
6. Finally, the hardware breakpoints can be configured.

This patching process is integrated in a single, low-priority task. The task is interruptible at any time and can be scheduled by the operating system. The operating system ensures that all deadlines are met, and interrupts the patching process when required. All critical sections during the HERA hotpatching process are atomic assembly instructions and hence cannot be interrupted by the operating system. In this way, HERA does not influence critical real-time tasks.

6.5.2 HERA Patch Development

Hotpatching requires a dedicated patch format. As discussed in Section 6.1, firmware updates on embedded devices are typically large due to their monolithic software architecture. However, HERA hotpatches must be small so that they can be applied during runtime. Furthermore, in addition to the actual patch, meta-information on processing and applying the patch needs to be provided. In the following, we describe how hotpatches for the HERA framework can be derived from traditional patches.

In HERA, we focus on patching security vulnerabilities. In contrast to feature updates, these patches are small and featureless. The application of security patches is time-critical to prevent exploitation of the patched vulnerabilities. So-called zero-day vulnerabilities are vulnerabilities that become public before patches are available. Furthermore, by making a patch available, also the vulnerability is made public so that exploitation is likely [183]. However, patch application comes at a risk: the patch could break the application. To prevent this, patches need to be tested, or even scheduled to maintenance windows, where interruption due to patch application is acceptable. Therefore, it is good practice in software development to separate security patches from feature updates to minimize the risk when applying time-critical patches [256]. This is similar to the motivation for hotpatching, which allows the application of patches without interrupting the patched service.

Hotpatches for HERA can be derived from traditional patches. So, when a traditional patch for a vulnerability is available, also a hotpatch can be developed without significant additional effort. To develop a hotpatch, the compiled version of the patched and unpatched firmware are compared using standard static code comparison techniques. In the case of security patches, this process is simple and straightforward. Popular tools for this task are for example IDA Pro[2] and Radare2[3]. These tools can also be used to determine the insertion points for the hotpatch as well as the location to store the hotpatch in the RAM. This approach to develop hotpatches also works when no source code of the firmware is available, as this is often the case for embedded and IoT devices. Furthermore, this step fits neatly into traditional software development processes. After a patch has been developed and tested, the hotpatch can be directly derived from the updated firmware without modifying the source code. So developers solely determine the changes on the assembly level and use this result to build the hotpatch, determining the patch insertion and storage locations. Developers may also perform changes to the patch, such as restoring stack values and registers. Currently, these changes are necessary due to the dispatcher that we implemented as on our prototype platform no direct jumps between ROM and RAM are possible. This is no limitation using HERA in practice. On other hardware platforms, these steps are not necessary. Furthermore, if a dispatcher is needed, in a full-featured updater these steps can be automatized as they are generic for every hotpatch.

6.5.3 Patch Development Guidelines

Summing up, to develop a new hotpatch to patch a vulnerability, the developer has to perform the following steps. We assume that there already exists a traditional patch that can be used to derive a hotpatch. Thus, the HERA hotpatching process fits in existing patch development processes.

1. Compile the patched version of the application.

[2]`https://www.hex-rays.com/products/ida/`
[3]`https://rada.re/n/`

2. Compare the original firmware with the patched version on assembly level using binary diffing techniques.
3. Extract the differences between both files. If needed, add modifications or further instructions.
4. Determine the breakpoint where to insert the hotpatch in the original application.
5. Combine the patch and this information into an update package.

The compilation in the first step can be done using the normal development workflow. The operations on the binary files can be done with standard tools such as Radare2 or IDA Pro. Finally, the resulting update package can then be tested and finally deployed to the vulnerable devices.

Example: Developing a Hotpatch for FreeRTOS

As an example, we show how we developed a hotpatch for a real-world vulnerability of the popular FreeRTOS real-time operating system. We followed the patch development guidelines in Section 6.5.3.

CVE-2018-16601 [194] describes a vulnerability in FreeRTOS that affects up to version 10.0.1. In these versions, the TCP/IP stack had a flaw that could cause memory corruption, allowing remote attackers to execute code or conduct a denial-of-service (DoS) attack that could disable the device. This attack works via a remote connection and does not require authentication. Therefore, the severity of this vulnerability was rated high with a CVSSv3 score of 8.1. Hence, it was urgent to patch this error to prevent exploitation, but updating a device would disrupt its service. So in practice, the patch application is delayed, even if the patches are highly critical. Therefore, we followed our previously outlined guidelines and developed a hotpatch that could function without interfering with the normal operation of the device.

The root cause of this buffer overwrite vulnerability was a lack of boundary checks, as the size of the IP header was not verified at any point [154]. So, crafting a special IP packet made it possible to write outside the reserved memory location. As FreeRTOS is an open-source application, its source code is freely available. Therefore, we compare the vulnerable version 9.0.0 with the current version 10.0.3, which gives us the mitigation for this vulnerability. In the file FreeRTOS_IP.c in function prvProcessIPPacket a check of the size of the IP header is added. The source code for this patch is shown in Listing 6.1. In the following, we show how we developed the hotpatch for this vulnerability. We backported the mitigation for CVE-2018-16601 from version 10.0.3 to version 9.0.0. Then, we compiled the original and the patched versions of FreeRTOS 9.0.0. Next, we used Radare2 to obtain a diff of the two binaries

Listing 6.1: Source code of the patch for CVE-2018-16601

```
if ( (uxHeaderLength >
     (pxNetworkBuffer->xDataLength
     - ipSIZE_OF_ETH_HEADER ))
     (uxHeaderLength < ipSIZE_OF_IPv4_HEADER))
{ return eReleaseBuffer; }
```

Listing 6.2: Assembly code of the patch for CVE-2018-16601

```
 1  lsls r3, r3, 2
 2  and r3, r3, 0x3c
 3  str r3, [r7, 0x24]
 4  + ldr r3, [r7]
 5  + ldr r3, [r3, 0x1c]
 6  + subs r3, 0xe
 7  + ldr r2, [r7, 0x24]
 8  + cmp r2, r3
 9  + bhi 0x801619a
10  + ldr r3, [r7, 0x24]
11  + cmp r3, 0x13
12  + bhi 0x801619e
13  + movs r3, 0
14  + b 0x80162ca
15  ldr r3, [r7, 4]
```

that resulted in the assembly instructions for the bounds check. Listing 6.2 shows these instructions. In a diff, a '+' or '-' symbol indicates that the line was added or removed. So, the hotpatch needs to add the lines 4–14 to implement the bounds check. Using this listing, we can also determine the insertion point for the trampoline to the hotpatch. To overcome the limitations of the FPB unit, the insertion point is moved to the first preceding instruction located at a 4-byte aligned memory address. Due to the system architecture, we needed some additional modification, as described in Section 6.5.2. If the patch proceeds directly after the trampoline, a return to the dispatcher and the 'jump_section' is necessary. However, if the branch target is beyond the patch, the offset between the RAM and ROM is usually too large to be included directly in a branch instruction. This control flow redirection involves manipulating the program counter register, which stores the memory location of the upcoming instruction to execute. So, it is the opposite case of the control flow redirect in the 'jump_section'.

6.5.4 Patch Application

Now that we have the source code of the hotpatch, we need to apply it on the device. This updater task is preemptable and runs at low priority so that the operating system schedules it without impairing time-critical tasks. The updater task performs the tasks described in Section 6.4.2. This includes downloading the patch, verifying its integrity, and extracting its contents. The patch consists of the actual patch, as well as additional meta-information required for patch application, such as the trampoline and the insertion point. Then the updater copies the hotpatch and the trampoline code to suitable memory locations and configures the FPB unit. Finally, the updater task triggers the atomic switchover. As the hotpatch activation is atomic, the patch can be activated anytime. When adapting this for practical usage, the patch can be a simple binary format or plain text files storing this information. Updating is a common feature of embedded and IoT devices [12]. The updater task can be integrated into an existing updater service, thereby reusing its methods to check for updates, downloading, and unpacking them. The patch format can also be reused. This way, HERA hotpatching can be integrated into existing systems and infrastructure.

Example: Application of Hotpatch for FreeRTOS.

For this case study, we implemented an updater service. First, the updater downloads the patch and verifies its integrity. Then, the updater copies the hotpatch and the trampoline to the correct memory locations. Afterwards, it configures the dispatcher and the FPB unit using the functions in the HERA library. Eventually, the updater enables the patch using the 'enable_single_patch' function. Now, when a breakpoint is hit, the FPB unit replaces the current instruction with a jump to the trampoline. Then, the hotpatch is executed. Afterwards, the control flow returns to the original location. Listing 6.3 shows the instruction replacement. The instruction in Line 4 is replaced with a branch-and-link instruction to the 'jump_section'. The 'jump_-section' directly modifies the program counter to redirect the control flow to the dispatcher. This section is necessary as no direct branch from ROM to RAM is possible due to architectural limitations. Listing 6.4 shows the assembly code of the dispatcher. The dispatcher verifies the source of each control flow redirection using the LR register. This register stores the return address, which is automatically saved by every branch-and-link (BL) instruction. This branch-and-link was previously inserted by the FPB unit. The dispatcher compares the current origin (LR) with all possible origins in the dispatcher to find the correct patch to execute.

Listing 6.3: Instruction replacement on breakpoint hit

```
push  {r7, lr}
[...]
ldrb  r3, [r3]
~~lsls  r3, r3, 2~~ bl jump_section
and   r3, r3, 0x3c
[...]
```

Listing 6.4: Dispatcher for hotpatching FreeRTOS

```
push {r3}
ldr.w r3, [0x20000052]; dispatcher entry
cmp lr, r3
beq 0x20000074; patch location
[...]
pop {r3}
pop {r7, pc}
```

6.6 Integration Guidelines

Integrating HERA hotpatching takes three straightforward steps.

1. Integrating the HERA library into the device to be patched.
2. Modifying the updater to handle HERA hotpatches according to Section 6.5.4.
3. Developing a suitable hotpatch according to the Patch Development Guidelines in Section 6.5.3.

After this, the newly developed hotpatch can be applied using the HERA hotpatching. In the following, we show how we integrated HERA into different real-world devices using these guidelines.

6.7 Evaluation

The HERA framework allows hotpatching of embedded systems. In contrast to traditional patching, HERA does directly influence the control flow of the patched system to apply patches during runtime. To show the practicability of this approach, we conduct an extensive case study. We implement HERA into two real-world embedded devices that operate under real-time constraints. We show how HERA can hotpatch vulnerabilities in these devices. Furthermore, we perform measurements to verify the timing behavior of the HERA patching process.

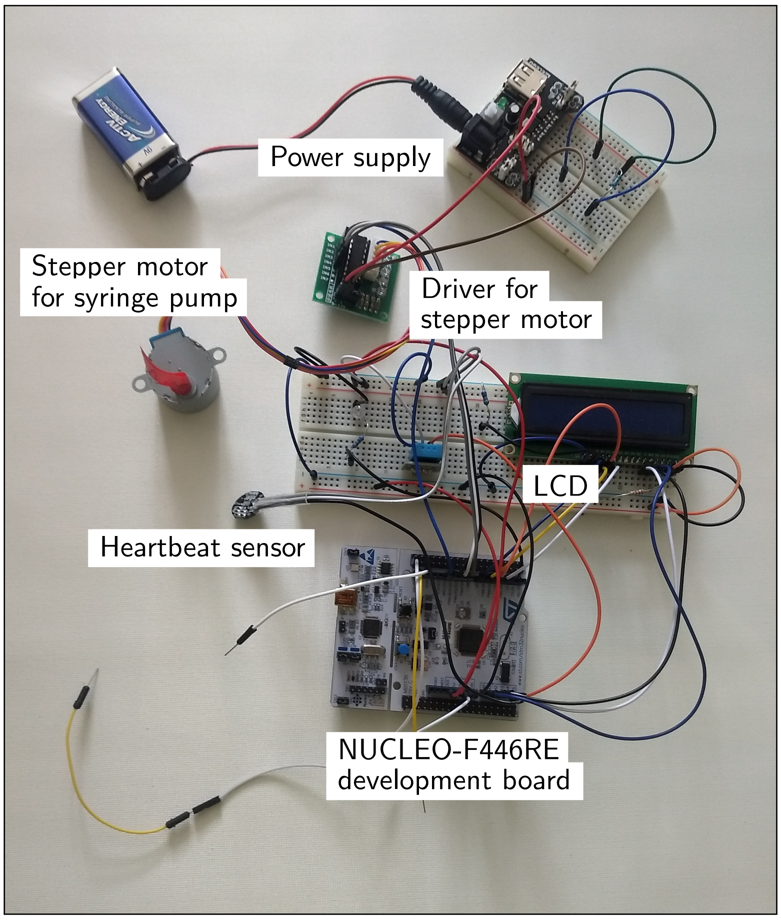

Power supply
Stepper motor for syringe pump
Driver for stepper motor
LCD
Heartbeat sensor
NUCLEO-F446RE development board

Figure 6.6: Photo of the implemented setup.

6.7.1 Implementation on Real-World Devices

We implemented HERA on two open-source real-world medical devices, a syringe pump, and a heartbeat sensor, using the STMicroelectronics NUCLEO-F446RE development board. In our setup, we also added the hardware peripherals such as the stepper motor for the syringe, the heart rate sensor, and the LC display to demonstrate the full functionality of these devices. Figure 6.6 shows the complete setup. Since there are no known vulnerabilities in the open-source implementation of both devices, we incorporated two typical memory corruption vulnerabilities that enable a remote attacker to compromise the devices. This is a common attack technique. The software on embedded devices is often programmed in memory-unsafe languages, just like the medical devices that we use for our example. Memory errors are widespread due to manual memory management, and they often result in severe software vulnerabilities [266]. Programming errors are among the most frequent causes of vulnerabilities with code execution capability for devices with local, remote, or internet access [211].

The vulnerabilities that we implemented allowed us to perform an out-of-bounds write due to improper input validation in the configuration interface. Using these vulnerabilities, we developed exploits for both devices using return-oriented programming (ROP) techniques [223]. Such out-of-bounds read and write vulnerabilities are typical for embedded devices. For instance, the aforementioned FreeRTOS CVE-2018-16601 allowed an out-of-bounds write. Another recent example is the Ripple20 vulnerabilities [152]. This is a series of vulnerabilities in a TCP/IP stack widely used in embedded devices. These vulnerabilities are also caused by improper memory handling or a lack of input validation. Ripple20 affected a wide range of different devices, including medical devices, industrial control systems, and home IoT devices, adding up to hundreds of millions of vulnerable devices.

Medical Devices

In the following, we describe the two medical devices, that we use in this case study. Commercial medical devices such as syringe pumps are usually costly. Hence, there is a growing interest in creating open-source alternatives that replicate such commercial medical devices [285]. This facilitates the production of such essential, potentially life-saving medical equipment, particularly in cases where industrial made medical devices are unavailable due to financial constraints or unforeseen events such as catastrophes. Both medical devices are based on the popular Arduino platform. The Arduino platform is widely used for open-source projects, due to its low cost and extensive software support [166]. Furthermore, the medical devices are well-suited candidates for our evaluation due to their stringent adherence to hard deadlines in order to maintain the patient's health and safety.

Syringe Pump. A syringe pump is a medical device that automatically injects medicine into a patient's body. Injecting the accurate amount of the substance, at the correct injection rate and precise time points is crucial. A malfunction that leads to incorrect dosing can have fatal consequences. As the syringe pump directly controls the stepper motor that performs the injection, a correct timing behavior is essential. We use the open syringe pump, an open-source implementation of a syringe pump. Abera et al. already used this project to demonstrate the feasibility of control flow attestation for embedded devices [3].

Heartbeat Sensor. The second device we will be discussing is a heartbeat sensor [151]. A heartbeat sensor is a device that measures a patient's heart rate. Accurate measurements require a precise timing. The sensor's real-time sampling rate is crucial in obtaining reliable measurements, as variations in timing can significantly impact the quality of the results. Consistent sampling is necessary to avoid unreliable measurements resulting from noise, spikes, or fluctuating signals. In addition to high availability, security is equally important. Malicious manipulations may lead to incorrect medication dosage or a false measurement of the heart rate. This can have fatal consequences for patients who rely on the accurate functioning of these devices such as in pacemakers.

We ported both applications from the Arduino platform to the ARM Cortex-M4 platform for our case study. While the original applications work on bare metal, we adapted them to work on the FreeRTOS operating system. As eluded earlier, we added the peripherals required for both devices, so that we yielded fully functional prototype devices for further evaluation. The resulting setup is shown in Figure 6.6.

Exploit Development

Both devices have a buffer overwrite/read vulnerability due to a lack of bounds checking in the configuration interface. This configuration interface is reachable remotely to control the devices. In our scenario, an attacker can exploit a buffer-overwrite/read error through this interface. Configuration interfaces are a common feature of embedded and IoT devices [208]. As discussed in the threat model, we assume a remote attacker with no direct physical access. We use these vulnerabilities to start an attack using return-oriented programming (ROP). The Cortex-M4 microcontroller has a Harvard architecture, but also a unified memory space. This modified Harvard architecture has a bus architecture, which allows to perform data and instruction access in parallel. Consequently, the same memory space is utilized by program code, data, and peripherals [292]. Typically, a Harvard architecture prevents the direct insertion of code. However, since it is implemented only at the microarchitecture level, the shared memory space still permits conventional code injection [10], as well as return-oriented programming (ROP) [242]. So, we are still able to implement an attack using return-oriented programming (ROP) on these devices.

For each of the devices we developed ROP exploits to manipulate the normal operation of the devices. These can have fatal consequences: We manipulated the syringe pump in a way that it injects a large amount of medicine without showing it on its display. In the case of the heartbeat sensor, we altered its measurements to arbitrary values. These examples demonstrate the critical importance of securing these devices.

Patch Development

We developed patches for both previously described applications to address the vulnerabilities caused by missing bounds checks. The patches ensure that any message exceeding the buffer size is dropped, given the fact that the applications utilize commands of fixed length. The patch development process followed the guidelines outlined in Section 6.5.3. First, we patched the vulnerability by adding the missing bounds checks. Then, we generated a diff at the assembly level between the compiled unpatched and patched versions. Minor modifications were necessary to reconstruct the stack frame pointer and adjust the jump instructions.

Hotpatching

Now, we show how we use the updater task to apply the hotpatch. First, the updater task directly loads the patch into RAM. In our implementation, the press of a button starts the transfer to RAM and the patch activation.

Updater Task. The updater task implements the features described in Section 6.7.1. It is based on a preemptable task that can be interrupted by the scheduler of FreeRTOS at any time. First, the updater task receives the hotpatch and copies it to a dedicated patch region in RAM. The updater task then initializes the trampoline and configures the FPB unit. Eventually, the patch is activated by means of an atomic switch, which is a single assembly instruction.

Atomic Switch using the FPB. The atomic switch to activate the hotpatch is one of the main concepts in the HERA framework. This atomic switch consists of a single assembly instruction, namely a register write, and therefore cannot be interrupted. This way, the critical patch activation situation does not require dedicated handling to prevent inconsistencies, for example, using atomic sections, as known from classical concurrent programming. Disassembling the binaries from this case study confirms that this register is accessed only once with a store instruction.

Summing up, in this case study we showed that HERA can be used to apply security patches on real-world embedded devices that operate under strict timing constraints. Before the patch application, the devices were vulnerable due to a typical memory corruption bug. We showed how these vulnerabilities could be exploited using return-oriented programming techniques to make the devices malfunction. We developed hotpatches for these vulnerabilities. These were then applied using the HERA framework. We showed that these patches successfully mitigated the attack by resolving the underlying memory corruption bug.

6.7.2 Measurements of the Overhead

In this series of experiments, we measure the overhead induced by HERA hotpatching. Responsiveness is a functional requirement of real-time devices. Hence, incorrect timing behavior of such devices is akin to device failure. Therefore, it is crucial to determine the exact timing behavior.

Since trampoline insertion is done on-the-fly and entirely in hardware, the overhead incurred by adding more code is minimal. The patching method relies on a small number of assembly instructions that are directly inserted. Hence, there are no compilers or intermediate software layers that can interfere with the systems' behavior. However, as the jump target cannot be directly addressed within a jump instruction, a trampoline to the dispatcher code is necessary, which is an additional small block of assembly instructions.

At the system level, the patch is executed as any other code, scheduled by the real-time operating system. Hence, the code can also be preempted if necessary. The overhead resulting from the FPB unit is deterministic, limited to only the instruction fetch or literal load for the replaced instruction [25]. This predictability enables the patching of real-time critical code sections. Furthermore, as the replaced instructions are defined during development, the developer can accurately calculate the necessary time for instruction fetch by considering the microarchitecture and CPU frequency. Next, we will further investigate the time required for switches using the FPB unit.

In order to determine the exact overhead, we measured the exact switching time using an oscilloscope [4], and taking the processor clock as a reference. We use the two medical devices described in Section 6.5 for our measurements. For the external pins, we inserted triggers to identify which instruction is being executed. These general-purpose input/output pins allow direct interaction between the processor and the oscilloscope. The technical reference manual confirms that the GPIO bus is directly linked to the CPU [259]. We set the bus frequency of the external pins to the same speed as the CPU clock, to minimize interference and obtain constant switch times. This way, the delay to switch the pins is only caused by the necessary instructions to change the pin state without waiting for the next switch cycle. To identify the switching time of the GPIO pins, we performed reference measurements without any FPB actions for comparison. As GPIO switches are also individual assembly instructions, these actions should also have the same constant timing properties as the atomic switch and the redirect to the patch. We repeated all measurements five times to identify possible variances in the results. However, we did not measure any deviation in any case. This is not surprising because our patch strategy fully controls execution and relies on hardware features that are executed atomically. As a consequence, in the following evaluation, we do not give any numbers on variances.

[4]Siglent SDS1104X-E [248]

Case	Switch Time	Pin Overhead	Difference
Syringe pump	1.524 µs	1.288 µs	236 ns
Heartbeat sensor	1.524 µs	1.288 µs	236 ns

Table 6.1: Measurement of the atomic switch time.

Case	Duration	Pin Overhead	Difference
While-Loop	1.624 µs	1.384 µs	240 ns
Syringe pump	1.456 µs	1.26 µs	196 ns
Heartbeat sensor	1.476 µs	1.288 µs	188 ns

Table 6.2: Time to abort instructions and switch to the jump-section.

We focus on two aspects of the hotpatching system. Real-time hotpatching requires predictability and minimal overhead to meet the deadlines. First, the time it takes to activate a single patch without interruption, called atomic switching time. Second, the time required to abort the execution of the current instruction and then insert and execute the trampoline. This is the control flow redirection. We have already discussed the runtime of these functions in theory. Now, using the oscilloscope, we verify that these properties hold in practice.

Atomic Switch Time

The atomic switch time is the time required for the atomic switchover that activates a patch. This takes place after the hotpatch has been prepared by the updater task. After the atomic switchover, the FPB unit inserts the patch every time a breakpoint is hit. Table 6.1 shows the results of the measurements. We measured the same times in both use cases, the syringe pump and the heartbeat sensor. The switch time including the time overhead to switch the GPIO pins is 1.524 µs. GPIO overhead, that is the time to directly switch the pins without the atomic switch is 1.288 µs. From these measurements, we can calculate the duration of 236 ns for the atomic switch. With the CPU clocked at 42 MHz, we can calculate that the 5 switch instructions take less than 10 clock cycles ($236\,\text{ns} \cdot 42\,\text{MHz} \approx 10$ clock cycles), with only one of them needing to be atomic. We repeated these measurements five times, but did not observe any deviation. Therefore, we conclude that the time required to activate the patch is small and constant, regardless of the patch and the patched application.

Instruction	Cycles	Duration
NOP	1	1.644 µs
PUSH {lr}	2	1.644 µs
LDR	2	1.644 µs
B.n	2	1.644 µs
UDIV	2–12	1.644 µs

Table 6.3: Time required to abort different instructions.

Control Flow Redirection

The second important timing property is the time required for trampoline insertion. This involves the dynamic replacement of one instruction and the execution of the corresponding trampoline. According to the datasheet, the expected overhead is the time to abort a single instruction [25]. However, since direct measurements of single instructions are not feasible, we included the time required to jump to the trampoline's target address in the measurement. To achieve this, we added a function called 'jump_section' as the target of the trampoline. The measurement starts before the replaced instruction and ends within the 'jump_section' for both the syringe pump and the heartbeat sensor applications. Table 6.2 shows the results of this experiment.

Similar to the previous experiment, we perform each measurement with and without the switch to determine the pin overhead and repeat all measurements multiple times to determine possible variances. First, we measure the time to branch into the 'jump_section' and return directly using a while-loop. In total, this takes 1.624 µs. Without the jump, it takes 1.384 µs, hence the switch itself takes 240 ns. The switch is slightly faster in the two medical devices, where the switches take 196 ns in the case of the syringe pump and 188 ns in the case of the heartbeat sensor, respectively. This is within our expectations, as the measurements of the while-loop include the instructions to return to the trampoline. Our ARM Cortex-M4 processor is clocked at 42 MHz. We can use this number to determine the number of clock cycles the switch requires. The while-loop takes $240\,\text{ns} \cdot 42\,\text{MHz} \approx 10$ cycles. The syringe pump is faster and takes only $196\,\text{ns} \cdot 42\,\text{MHz} \approx 8.2$ cycles, and the heartbeat sensor takes $188\,\text{ns} \cdot 42\,\text{MHz} \approx 7.9$ cycles. However, since instructions take at minimal one cycle to execute [25], a difference of only two cycles is minimal. Overall, the measured times suggest that the overhead for the control flow redirection is negligible.

Instruction Abort

Next, we investigate the overhead due to the instruction abort. Upon a breakpoint hit, the FPB unit aborts and replaces the current instruction. This results in an

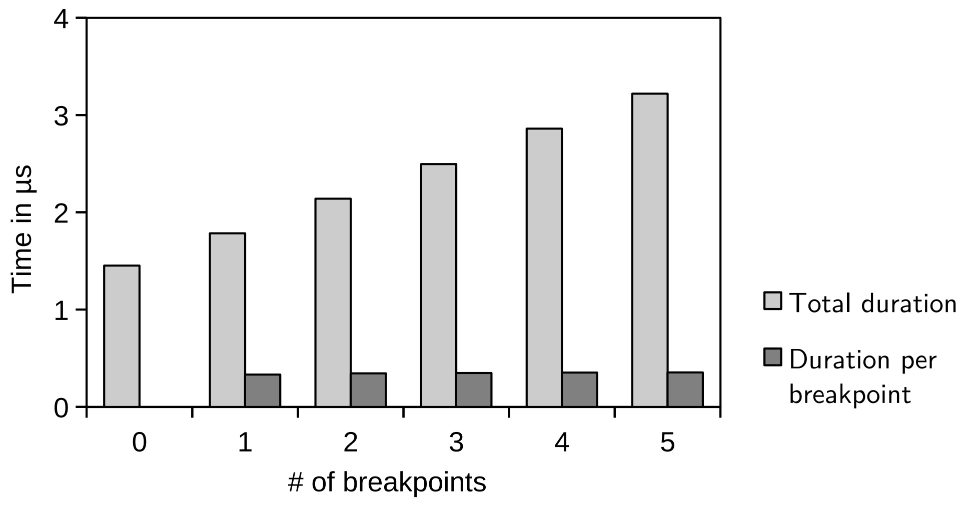


Figure 6.7: The duration of the patching depends on the number of breakpoints.

additional instruction fetch. While instructions may take multiple processor cycles to execute [25], this instruction fetch takes constant time and is independent of the execution time of the aborted instruction. We verify this in an experiment using different instructions. The results are shown in Table 6.3, the numbers of CPU cycles are taken from the reference manual [25]. The selected instructions are common and represent a broad range of execution costs. The no-operation (NOP) instruction requires a single CPU cycle, the PUSH instruction takes two cycles, and division can take up to 12 cycles to complete. Notably, the execution time has not changed when exchanging different instructions. Independent of the instruction, the time to abort the instruction is 1.384 µs including the pin overhead. For this measurement setup, we determined the pin overhead to remain constant at 1.384 µs. Thus, it can be concluded that the transaction abort is constant and independent of the replaced instruction.

6.7.3 Further Measurements

We performed two additional experiments. First, we verify the timing behavior of HERA with multiple active breakpoints. Furthermore, we performed a complete end-to-end example in which we monitored the operation of the syringe pump during hotpatch application.

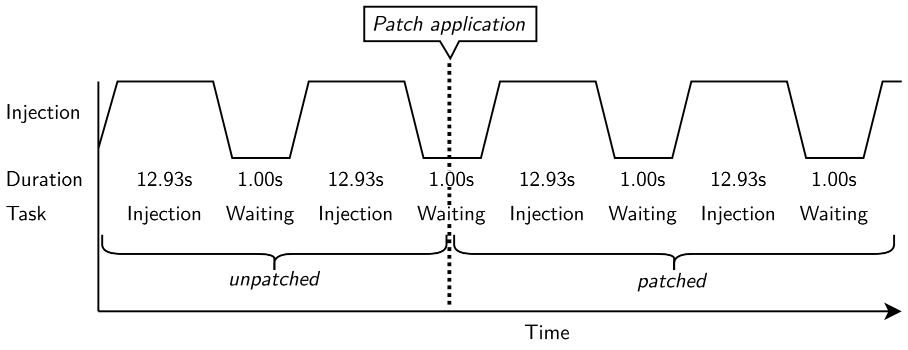


Figure 6.8: Measurements of the full end-to-end experiment on the syringe pump.

Multiple Breakpoints

In this experiment, we measure the overhead of multiple active breakpoints. Using HERA it is possible to have multiple active breakpoints simultaneously. First, it may be necessary to patch more than one vulnerability. Second, hotpatches may consist of more than one part, thereby requiring multiple breakpoints to activate the patch correctly. Using HERA, this is straightforward to implement by using multiple trampolines. The only limitation is the number of physically available breakpoints in the FPB unit. The ARM Cortex-M4 processor supports up to six breakpoints [25]. However, in our experiment, we were only able to evaluate up to five simultaneous breakpoints due to the requirement of one breakpoint for the measurement setup. The setup involved a while-loop containing NOP instructions, which were replaced step-by-step with trampolines by configuring the FPB breakpoints. Furthermore, we added one case with zero breakpoints to obtain a reference baseline, indicating the overhead due to pin triggering. The measured times for each number of breakpoints are shown in Figure 6.7. Using the baseline measurement, we were able to determine the duration per breakpoint for all numbers of breakpoints. We repeated each measurement five times, and as expected, we did not observe any deviation as the FPB unit switches are atomic instructions with a fixed execution time. It is safe to conclude that the duration per breakpoint is constant, and multiple trampolines can be inserted with a known and fixed overhead.

End-to-End Case Study

To show the full functionality of HERA hotpatching practice, we conducted a full end-to-end experiment using the syringe pump. Using an oscilloscope, we monitored the operation of the syringe pump before, during, and after the hotpatching process. The results are shown in Figure 6.8. In 1 s intervals, the syringe pump injects 1 mL.

We first measured the operation of the unpatched syringe pump. Then, we applied and activated the patch. The operation continued without interruption. With the oscilloscope, we verified that the timings were not impacted by the patching process or the patch application by monitoring the process continuously. The patch did not cause any delay and the syringe pump remained fully functional during the hotpatching process.

6.7.4 Summary

Summing up, our case studies on real-world devices demonstrate that HERA provides an efficient and effective mechanism to hotpatch resource-constrained embedded devices with real-time constraints. Through the case study of medical devices, we have demonstrated that HERA can be used to perform hotpatching on critical embedded systems. Furthermore, we used HERA to hotpatch CVE-2018-16691, a vulnerability in the popular FreeRTOS, thereby showing how also real-world vulnerabilities can be patched. In the evaluation, we used HERA to patch two real-world medical devices. We verified the timing behavior of HERA in a comprehensive measurement study. The measurements show that the performance overhead induced by HERA is negligible both in theory and practice. The control flow redirection overhead is on a sub-instructional level, while the time for the atomic switch is only a single assembly instruction that is either executed completely or not executed at all. The trampolines are designed with the minimum number of instructions required to jump to the patch, and it is possible to insert multiple trampolines with predictable and negligible overhead, as verified by a full end-to-end experiment. Therefore, we conclude that HERA is highly practical for patching systems with hard real-time properties, which have the most strict requirements.

6.8 Security Discussion

The novel approach of HERA allows practical hotpatching of embedded devices with real-time constraints. However, several security aspects have to be considered when using HERA.

Attacks on the Updater Service. As in any patching scheme, in HERA hotpatching the updater service inhibits a crucial role. The updater service by design is able to modify the system in order to apply the patch. Hence, a malicious or compromised updater service is able to modify the system and thereby further compromise it. The patched system cannot prevent this. Therefore, the security of the updater service is of utmost importance for HERA hotpatching as well as in any other patching process.

Updater services are a standard feature of embedded systems and hence widespread. Therefore, we assume the updater service to be secure. In addition, before applying the update, we assume that the updater service checks the integrity and authenticity of the update package using cryptographic functions.

Security of Patches. By design, patches cause changes in the patched applications. As HERA is able to reuse existing patches as hotpatches, the application of hotpatches using the HERA framework does not cause additional security issues compared to traditional patching. The updater service verifies the integrity and authenticity of the hotpatch, typically using hash sums and digital signatures so that an adversary cannot modify existing hotpatches as any changes to the hotpatches will be detected by the updater service.

Side-effects on Normal Operation. When applying hotpatches, the patch may not interfere with the normal operation of the patched system. In particular, in real-time systems, the patch may not cause timing violations of real-time tasks. As HERA uses a hardware feature to guarantee patch activation within one processor instruction, the patch activation process is fully deterministic. The preparation of the patch is done before patch activation in a low-priority background task that does not interfere with normal operation. Hence, HERA hotpatching does not cause any unplanned side-effects or interdependencies with the patched system and is therefore also suitable for highly critical systems with hard real-time and safety constraints.

Exploitation of HERA. To implement hotpatching, HERA introduces a dedicated patch region to store hotpatches. By design, this memory region has to be writable and executable during runtime and can therefore be used by an attacker to load and execute malicious code. To prevent this, the critical memory area can be protected.

Embedded devices often feature memory management techniques such as memory protection units (MPUs) or the more sophisticated memory management units (MMUs). MPUs are a widespread feature of embedded devices and can be used to implement permission control on memory. MPUs allow splitting the memory into different regions, each with its own set of access permissions. The MPU then enforces these memory protection rules on these regions. This way, MPUs can prevent unauthorized access or modifications of the memory, for instance, to protect memory areas that store sensitive information from being read by unauthorized processes.

The ARM Cortex-M also features such an MPU. This MPU can be used to limit access to the patch region to the updater service, so that other processes cannot exploit this memory area for malicious purposes.

Furthermore, HERA does not offer any external interfaces besides the updater service. The updater service is assumed to be secure and checks the integrity and authenticity of patches prior to application. Therefore, we conclude that HERA hotpatching does not increase the attack surface of the system.

Summary. Summing up, HERA enhances embedded devices with real-time constraints with hotpatching capability without increasing their attack surface compared to traditional patching. Hotpatches can be derived from traditional patches on a binary level. The updater service for HERA verifies the integrity of the hotpatch similar to normal patches. The patch region that HERA uses for the hotpatch can be secured for example using an MPU. The HERA hotpatching approach prevents interdependencies with the normal operation of the patched system. This makes HERA also suitable for systems with real-time constraints.

6.9 Related Work: Hotpatching

The motivation for utilizing hotpatching arises from the need to fulfill high availability requirements. The concept of hotpatching or dynamic software updates, thus patching software during runtime, is an old idea and has been primarily directed towards conventional software [238]. Hotpatching is popular for server applications that need to run continuously [63, 122, 213]. In traditional patching, services need to be restarted, resulting in a downtime of the patched system. This downtime must be dealt with, for instance, by scheduling patches, planning maintenance windows, or switching over to secondary instances. The different hotpatching methods can be classified based on the model used and the impact of the software update [123].

Javelus is a modified Java virtual machine that reduces the latency when dynamically applying updates on Java applications. They proposed a lazy update approach, where objects are only updated when they are used, thereby reducing the latency when applying the patch during runtime [128]. There are multiple approaches to implementing hotpatching on Android devices. KARMA allows live patching of Android kernels to fix security vulnerabilities on Android systems without having to wait for official updates from the vendor [66]. InstaGuard extends this approach and allows applying generic rule-based patches instead of executable code [65]. Vulmet is a tool to automatically generate hotpatches for Android devices [290].

Traditional updates, which typically involve restarting programs or systems, erase the system's current state. However, hotpatching requires to maintain this state. Therefore, when developing hotpatching schemes, one of the main challenges is implementing changes while also preserving and transferring the current system state [132]. This is especially challenging in concurrent systems. Many applications run concurrent threads, for instance, server applications. Hotpatching such applications poses additional challenges due to the complexity of their states [225].

UpStare is a hotpatching framework for server applications that uses a special stack reconstruction algorithm to update active functions [176]. Another approach is to bring the application to a defined state before applying the update. The Kitsune framework uses so-called update points to apply hotpatches. However, programmers have to determine these update points manually [136]. In contrast, Giuffrida et al. propose an automated approach that synchronizes the states of processes over time to apply hotpatches instead of focusing on a single specific point in time [122]. The POLUS framework also performs state synchronization to automatically update server applications without update points. POLUS involves running the new and old data structures side by side, while gradually transitioning to the new structure through synchronization [63].

However, not all hotpatching schemes need to incorporate a complex state transfer. Although standard software updates can be arbitrarily complex, replacing large parts of the application or even introducing new functionality, security patches are typically “small, isolated and featureless” [14]. Thus, a popular approach to implement hotpatching is using relocatable executables [136, 150, 219].

With the upcoming of the Internet of Things, thus embedded devices enhanced with Internet connectivity, patching and hotpatching of these devices became important to address security issues. Moreover, as eluded earlier, there are many embedded devices that have high availability constraints. On these devices hotpatching is the only solution to apply timely updates during runtime. Felser et al. presented an approach to implement hotpatching on sensor nodes using a special architecture. Patching sensor nodes is challenging because of their limited computational resources. The process automatically calculates the differences and creates an image to incrementally link new code to the existing application [109]. In sensor nodes, the need for low energy consumption often also is a significant limitation. Zhang et al. propose a hotpatching scheme for energy-harvesting embedded devices that performs in-place replacement using trampolines [303]. Cetratus is a hotpatching framework for safety-critical systems, such as in the industrial Internet of Things. This framework introduces a quarantine mode to set up and monitor hotpatches [190]. The Cetratus framework was also extended for the usage in the energy management of smart cities [189]. Another hotpatching framework is Piston. This is a framework to hotpatch devices that were not designed for hotpatching by exploiting the devices and then modifying their code on-the-fly [231].

Hotpatching of real-time systems presents additional challenges [278, 279, 280]. Wahler, Richter, and Oriol presented a framework that allows for hotpatching of real-time systems entirely in software. The update process builds upon update points. These are points in time that are suitable to apply hotpatches. However, they focus on systems with a cyclic architecture, such as control systems that base upon a control loop or systems with measurement tasks. In addition, they assume that the update fits in one cycle. Furthermore, the critical update process needs to take a linear amount of time. To overcome these limitations, Wahler et al. propose a state synchronization algorithm to synchronize old and new components, allowing for updates with arbitrarily large states [279]. For practical usage, this component-based hotpatching model has been extended, verified, and integrated into FASA framework (Future Automation System Architecture) [278].

Another crucial aspect when applying patches during runtime is fault tolerance. In high-availability systems, fault tolerance is often as important as availability. Replication or redundancy is a common approach to achieve fault tolerance, and several replication schemes can also be used for applications with hard real-time constraints [68]. Implementing hotpatching on replicated systems is often straightforward, as such systems can be patched one at a time, as they usually substitute each other during downtime. Although there are multiple solutions to achieve replication [130], redundancy is not always a practical solution, as it is costly and requires hardware and resources that are not present in embedded systems with minimal hardware. Especially for enhancing legacy embedded devices, such schemes are not an option. Thus, direct hotpatching remains the only practical solution for these systems.

6.10 Summary and Conclusions

In this chapter, we presented HERA, a novel framework for hotpatching legacy embedded devices under hard real-time constraints. To date, the challenges of applying hotpatches on embedded devices that have low resource availability due to the need for low cost and low energy consumption, guaranteed availability, and hard real-time capabilities have not been addressed by current research. HERA uses hardware debugging features of commercial off-the-shelf hardware to activate patches within one processor instruction, thereby allowing an atomic switchover that cannot be interrupted, minimizing the time overhead when patching, and guaranteeing deterministic behavior and timing. The prior preparation of the patch was done during idle time, without interfering with the normal operation of the patched device. In summary, HERA is able to apply patches with negligible and predictable overhead.

In the evaluation, we verified the behavior and showed the practicability of HERA. In a case study, we integrated HERA into two real-world medical devices, a syringe pump, and a heartbeat sensor, using the flash patch and breakpoint unit of a standard ARM Cortex-M4 processor. Furthermore, we used HERA to patch a real-world vulnerability in the widely used real-time operating system FreeRTOS, demonstrating the effective and efficient applicability of HERA to hotpatch real-time critical embedded systems. In our implementation guidelines, we explain the usage of the HERA framework and the development of suitable hotpatches. We conducted an extensive measurement study to determine the overhead due to hotpatching. Using an oscilloscope, we verified that HERA actually activates patches with predictable and negligible overhead. In a full end-to-end example, we verified the functionality of HERA by patching a vulnerability and verifying that it has been correctly patched while monitoring the timing behavior of the patched device. This showed that HERA is a practical solution to hotpatch even the most critical embedded devices that operate under hard real-time constraints.

HERA, by design, is only able to reliably patch uncompromised systems. The attacker can stop or manipulate the patching process on a compromised system without being detected. Remote attestation on the other hand allows reliable detection of compromises even on fully compromised systems. A combination of hotpatching and remote attestation would allow verifying whether hotpatches have been successfully applied, even on untrusted systems.

CHAPTER 7

Summary and Conclusions

In the modern world, more and more functions are relying on embedded devices. These embedded devices ease daily life, for example, in the manifold Internet of Things (IoT), but also often take over critical functions, for example, in vehicles, industrial machinery, or medical devices. Despite their criticality, they suffer from the same security vulnerabilities as commodity computer systems while at the same time having less security features and significant more technical restrictions. A solution to address vulnerabilities is to replace or patch these devices. However, a replacement is in most cases not feasible because IoT devices are often deeply integrated into other systems or consist of specialized hardware. Alternatively, software-based security solutions do not require any hardware modifications and, therefore, are a worthwhile solution for dealing with the security issues of such legacy embedded devices. This dissertation shows different software-based approaches to enhancing existing devices with new security features based on remote attestation without modifying the existing hardware. As remote attestation can only detect but not prevent a compromise, in addition, we propose a novel, interruption-free hotpatching framework, since patching is the only solution to solve the underlying root cause for a compromise.

Certainly, there is no universal solution to integrate remote attestation into existing devices due to the complex nature of the problem and the diversity of the embedded devices. Therefore, we propose different solutions to common problems. This dissertation shows how typical features of existing devices can be reused to enhance security and enable remote attestation. The focus is on practical solutions that can be used in practice. For each solution, we explained in detail the prerequisites for implementation and how it can be integrated into existing devices. Furthermore, we showed in real-world examples the applicability of the proposed solution to demonstrate the full functionality and practicability.

7.1 Dissertation Summary

In Chapter 1 we motivated the use of remote attestation and gave an overview of this dissertation. In the background in Chapter 2 we provided structured information on different types of embedded devices, especially real-time systems, which are the most critical class of such devices. Then, we gave an overview of different attestation schemes, focusing on the differences in their architectures and security guarantees.

In Chapter 3, we presented RealSWATT, the first software-based attestation scheme that is also applicable to devices with real-time constraints. It utilizes a dedicated processor core and an IoT gateway to allow secure software-based attestation in standard IoT settings, such as wireless networks of IoT devices. This is achieved by introducing continuous attestation, a concept where devices are continuously attested in the background. This allows compromising between the attestation interval and the communication overhead between the verifier and the attested device.

SCAtt-man, presented in Chapter 4 provides user-understandable remote attestation, while solving the root of trust problem in software-based attestation. This is achieved by using side-channels for the attestation that users can observe. This way, users can identify the device that is currently being attested. We integrated SCAtt-man into a smart speaker that users can attest with an Android app on their smartphone. The communication was performed via audible sound. This concept represents a realistic and user-friendly setting. For the attestation, we implemented a data-over-sound protocol and optimized its reliability to ensure a dependable attestation. In a user study, we found that SCAtt-man is easy to use and that users would use it in practice if their devices featured such an attestation method.

In Chapter 5, we presented DMA'n'Play, which uses direct memory access to perform remote attestation of embedded devices. With DMA'n'Play, the verifier can directly monitor the memory of the attested device and detect manipulations. In contrast to traditional remote attestation schemes, this approach eases the implementation of the verifier logic and allows the integration of complex checks of the attested device. We achieve this by connecting a tiny embedded device to the attested system, which can directly monitor the attested device's memory. This low-cost device can then either relay the data from the attested device or take over the role of the verifier. In order to do so, the DMA controller of the attested device then directly sends the memory content to the verifier via a serial connection. Hence, to implement DMA'n'Play, only the configuration of the DMA controller has to be adapted. This architecture makes DMA'n'Play especially suitable for embedded devices in which the software cannot be altered.

Chapter 6 presents HERA, a hotpatching framework for devices under real-time constraints. HERA allows to hotpatch embedded devices with minimal timing overhead. This hotpatching is achieved by leveraging common hardware debugging features to apply patches, thereby providing exact predictability. Thus, even devices under hard real-time constraints can be patched during normal operation. We verified this behavior in an extensive measurement study. While we implemented HERA using the flash patch and breakpoint unit of an ARM Cortex-M3/4 processor, such a breakpoint functionality is in fact a widespread feature of commodity processors for embedded systems.

To show the broad applicability, all of these new solutions have been integrated into adequate applications and devices, such as medical systems, drones, and IoT devices. Detailed descriptions of the implementation show how we integrated these new security solutions into existing devices. Integration guidelines give a quick overview of how to implement these techniques into other devices and applications, focusing on legacy devices where no source code is available. When integrated into systems with real-time constraints, additional measurements and experiments showed that the timing behavior was not impaired and that the timing constraints were still met.

In summary, this dissertation presents four new, different, and entirely software-based security solutions for embedded devices. They can also be used on existing legacy devices, as they do not require any hardware modifications. These solutions are specifically designed to tackle shortcomings of existing approaches and provide practical solutions to real-world problems, thereby enhancing the security of embedded devices.

7.2 Comparing Remote Attestation with Other Security Enhancements

There exist many security enhancements to hinder the exploitation of vulnerabilities, such as ASLR, non-executable memory, control flow integrity, and stack canaries. All of these techniques can only be implemented on the device itself. Remote attestation by design requires an external entity to verify the attestation reports of the monitored device. This aspect has a significant impact on the practical application of remote attestation. Without a verifier, remote attestation does not give any security guarantees. The necessity of a remote verifier can cause scalability issues when applying remote attestation to large fleets of devices. Depending on the attestation scheme and the complexity of the attested device, the computing power required for the verifier may be significant: While static attestation of a device's firmware is straightforward to verify as attestation reports are simple and stateless, extensive configuration and runtime properties are complex to verify due to the large state space. In devices with many valid states the verification of attestation reports becomes a complex task requiring significant resources, especially when the validity checks require state information of every individual attested device. Furthermore, remote attestation requires continuous communication with a verifier to verify the attestation reports.

On the one hand, these requirements impose significant limitations on the practical applicability of remote attestation. However, on the other hand, remote attestation also offers unique advantages. With remote attestation, security enhancements can be implemented at a central instance without altering the targeted device. Furthermore, remote attestation by design provides a monitoring solution that allows one to continuously check a device's state and detect attacks, causing deviations from normal behavior. Usually, new security techniques always need to be implemented directly on the device. With remote attestation, these checks are instead implemented at the verifier, making it possible to add or change these checks without changing the targeted device. This aspect is especially relevant for devices that are hard to change, such as embedded devices that are deeply integrated into other systems or real-time systems with strict timing constraints. So there is no need to change the attested device to integrate additional checks or security mitigations. The inherent communication to the verifier in remote attestation schemes has significant advantages in systems with central management, as it inherently allows monitoring of the attested devices. This makes it possible to monitor devices and their behavior, thereby detecting compromises and other deviations from expected states and behavior.

In practice, monitoring and verification functionality can be integrated into existing devices and hence does not require new, dedicated verifier devices. For instance, verifier logic can be integrated into existing network components, like the configuration manager, as proposed in RealSWATT in Chapter 3. Furthermore, remote attestation allows for building trust in remote systems by checking integrity. Devices often operate in untrusted environments. Embedded and IoT devices are deployed in homes, factories, or vehicles. In contrast to security enhancements that are only working on a single device, remote attestation sends self-measurements to a remote verifier. When attackers circumvent certain security mitigations, such as ASLR, they receive direct feedback when their exploit is working. In contrast, in remote attestation, the attackers are not aware of the exact checks and are not able to determine whether the compromise is detected.

In conclusion, in the first place, developing secure software and patching vulnerabilities in a timely manner is preferable over mitigations or detection techniques such as remote attestation. Remote Attestation, however, is a viable solution for specific use cases, such as highly critical systems, where a compromise can have severe consequences, or to enhance trust in remote devices by checking their integrity.

7.3 Future Research Directions

In this thesis, we developed remote attestation techniques for legacy embedded devices. In addition to the proposed solutions, we identified the following worthwhile directions for future research.

Software-Based Runtime Attestation of Real-Time Systems. Integrating runtime attestation into any existing application inherently changes its execution and timing behavior due to the necessary measurements. Even integrating instrumentation into an existing application changes its timing behavior. This change in timing behavior is especially relevant for real-time applications that have been developed to meet strict timing deadlines. Changing the timing behavior requires extensive checks, which is impractical for legacy devices. How to develop an attestation solution that solves this fundamental problem and allows runtime attestation of existing devices without negatively impacting their timing behavior is still an open problem.

Attestation of Complex Software. In this thesis, the focus is on small, embedded devices with a relatively small code base. However, the code size of embedded devices, as well as their computational capabilities, tend to increase. When extending attestation on larger, more complex systems and including dynamic properties like control flow or data, then the resulting complexity and frequent attestation tasks will drastically reduce the performance when using traditional approaches. Therefore, these systems demand for more sophisticated approaches to implement practical remote attestation.

Efficient Verification. Until now, little attention has been paid to the implementation of the verifier. Usually, it is assumed that the verifier has unlimited resources. Therefore, the verifier can have prior knowledge of all benign states of the attested system and check the attestation reports at any rate. Although this is a reasonable assumption in academia, this is a significant limitation for practical usage.

Especially when attesting runtime properties and frequently changed data and configuration, the complexity of the verifier must be considered. For example, complex checks, large databases, or extensive state exploration, even upon small changes to the attested system, make remote attestation impractical. The effort and computing power of the verifier must be in proportion to the attested system and the security benefits that the remote attestation gives. This is particularly relevant considering large-scale attestation, which includes both systems with a large software base, as well as large networks of devices, such as in autonomous vehicles, IoT systems, or medical devices. Therefore, it is a worthwhile goal to investigate and optimize the efficiency of verifiers. This includes both the runtime as well as the initial effort to compile the information required for the attestation.

Usability Aspects of Attestation. Another aspect which has paid Little attention to are the usability aspects of remote attestation. In Chapter 4, we investigated the usability of an attestation scheme and found that users generally trust and would use it. However, we did not consider important aspects. For instance, it remains an open question of how to resolve a compromise that has been detected by remote attestation. In addition, a restoration process for a compromised device is yet to be designed. Furthermore, it is unclear whether users trust a previously compromised device that is proven benign again by attestation.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. “Control-flow Integrity Principles, Implementations, and Applications”. In: *ACM Transactions on Information and System Security (TISSEC)* 13.1 (2009), pp. 1–40. DOI: 10.1145/1609956.1609960.

[2] Ali Abbasi, Jos Wetzels, Thorsten Holz, and Sandro Etalle. “Challenges in Designing Exploit Mitigations for Deeply Embedded Systems”. In: *Proceedings of the 2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2019, pp. 31–46. DOI: 10.1109/EuroSP.2019.00013.

[3] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. “C-FLAT: Control-Flow Attestation for Embedded Systems Software”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS 16)*. ACM, 2016, pp. 743–754. DOI: 10.1145/2976749.2978358.

[4] Tigist Abera, Raad Bahmani, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Matthias Schunter. “DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous Systems”. In: *Proceedings of the 2019 Network and Distributed Systems Security Symposium (NDSS 19)*. Internet Society. 2019.

[5] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. “Intel Virtualization Technology for Directed I/O”. In: *Intel Technology Journal* 10.3 (2006), pp. 179–192. ISSN: 1535864X.

[6] Abbas Acar, Hossein Fereidooni, Tigist Abera, Amit Kumar Sikder, Markus Miettinen, Hidayet Aksu, Mauro Conti, Ahmad-Reza Sadeghi, and Selcuk Uluagac. “Peek-a-boo: I see your smart home activities, even encrypted!” In: *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 2020. DOI: 10.1145/3395351.3399421.

[7] Airbus. *Operating life*. 2022. URL: https://www.airbus.com/en/products-services/commercial-aircraft/the-life-cycle-of-an-aircraft/operating-life (visited on 08/31/2022).

[8] Panagiotis Aivaliotis, Z Arkouli, Konstantinos Georgoulias, and Sotiris Makris. “Degradation curves integration in physics-based models: Towards the predictive maintenance of industrial robots”. In: *Robotics and Computer-Integrated Manufacturing* 71 (2021). ISSN: 0736-5845. DOI: `10.1016/j.rcim.2021.102177`.

[9] Samer Al-Kiswany, Abdullah Gharaibeh, Elizeu Santos-Neto, George Yuan, and Matei Ripeanu. “StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage systems”. In: *Proceedings of the 17th International Symposium on High-Performance Distributed Computing (HPDC-17 2008)*. ACM, 2008, pp. 165–174. DOI: `10.1145/1383422.1383443`.

[10] One Aleph. “Smashing the stack for fun and profit”. In: *Phrack Magazine* (1996). URL: `http://www.shmoo.com/phrack/Phrack49/p49-14`.

[11] Omar Alrawi, Charles Lever, Kevin Valakuzhy, Ryan Court, Kevin Z. Snow, Fabian Monrose, and Manos Antonakakis. “The Circle Of Life: A Large-Scale Study of The IoT Malware Lifecycle”. In: *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021. ISBN: 978-1-939133-24-3. URL: `https://www.usenix.org/conference/usenixsecurity21/presentation/alrawi-circle`.

[12] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. “SoK: Security Evaluation of Home-Based IoT Deployments”. In: *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 1362–1380. DOI: `10.1109/SP.2019.00013`.

[13] Riham Altawy and Amr M Youssef. “Security, Privacy, and Safety Aspects of Civilian Drones: A survey”. In: *ACM Transactions on Cyber-Physical Systems* 1.2 (2016). DOI: `10.1145/3001836`.

[14] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. “OPUS: Online Patches and Updates for Security”. In: *Proceedings of the USENIX Security Symposium (USENIX Security 05)*. USENIX Association, 2005. URL: `https://www.usenix.org/conference/14th-usenix-security-symposium/opus-online-patches-and-updates-security`.

[15] AMD. *AMD I/O Virtualization Technology (IOMMU) Specification*. 2021. URL: `https://www.amd.com/system/files/TechDocs/48882_IOMMU.pdf` (visited on 04/20/2022).

[16] Mahmoud Ammar and Bruno Crispo. “Verify&Revive: Secure Detection and Recovery of Compromised Low-End Embedded Devices”. In: *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. ACM, 2020, pp. 717–732. ISBN: 9781450388580. DOI: `10.1145/3427228.3427253`.

[17] Mahmoud Ammar, Bruno Crispo, Ivan De Oliveira Nunes, and Gene Tsudik. "Delegated Attestation: Scalable Remote Attestation of Commodity CPS by Blending Proofs of Execution with Software Attestation". In: *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 2021, pp. 37–47. ISBN: 9781450383493. DOI: `10.1145/3448300.3467818`.

[18] Mahmoud Ammar, Bruno Crispo, and Gene Tsudik. "SIMPLE: A Remote Attestation Approach for Resource-constrained IoT devices". In: *Proceedings of the 11th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS 2020)*. IEEE, 2020, pp. 247–258. DOI: `10.1109/ICCPS48487.2020.00036`.

[19] Mahmoud Ammar, Wilfried Daniels, Bruno Crispo, and Danny Hughes. "SPEED: Secure Provable Erasure for Class-1 IoT Devices". In: *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy (CODASPY 18)*. 2018, pp. 111–118. DOI: `10.1145/3176258.3176337`.

[20] Android Open Source Project. *A/B (Seamless) System Updates*. 2020. URL: `https://www.arm.com/products/silicon-ip-cpu` (visited on 07/09/2020).

[21] Anna-senpai. *GitHub - Mirai Source Code*. 2017. URL: `https://github.com/jgamblin/Mirai-Source-Code` (visited on 04/09/2021).

[22] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. "Understanding the Mirai Botnet". In: *Proceedings of the 26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 2017, pp. 1093–1110. ISBN: 978-1-931971-40-9. URL: `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis`.

[23] ARM Limited. *AMBA 3 AHB-Lite Protocol Specification*. 2006. URL: `https://www.eecs.umich.edu/courses/eecs373/readings/ARM_IHI0033A_AMBA_AHB-Lite_SPEC.pdf` (visited on 11/21/2022).

[24] ARM Limited. *Security Technology Building a Secure System Using TrustZone Technology (White Paper)*. 2009. URL: `https://documentation-service.arm.com/static/5f212796500e883ab8e74531` (visited on 07/20/2023).

[25] ARM Limited. "Cortex-M4 Processor Technical Reference Manual". In: *Revision: r0p1, ARM 100166_0001_00_en* (2015). URL: `https://static.docs.arm.com/100166/0001/arm_cortexm4_processor_trm_100166_0001_00_en.pdf` (visited on 12/16/2020).

[26] ARM Limited. *ARM System Memory Management Unit Architecture Specification.* 2016. URL: `https://documentation-service.arm.com/static/5f900d34f86e16515cdc08fb` (visited on 04/20/2022).

[27] ARM Limited. *TrustZone Technology for Armv8-M Architecture.* 2018. URL: `https://developer.arm.com/documentation/100690/latest/` (visited on 04/19/2022).

[28] ARM Limited. *TrustZone for Armv8-A.* 2019. URL: `https://documentation-service.arm.com/static/602167b6873dd96c4deaf49b` (visited on 04/19/2022).

[29] ARM Limited. *Arm Cortex-M4 Processor Technical Reference Manual.* 2020. URL: `https://developer.arm.com/documentation/100166/0001` (visited on 04/19/2022).

[30] ARM Limited. *Arm Processors for the Widest Range of Devices–from Sensors to Servers.* 2020. URL: `https://www.arm.com/products/silicon-ip-cpu` (visited on 07/08/2020).

[31] ARM Limited. *Configuring and enabling the MMU.* 2022. URL: `https://developer.arm.com/documentation/den0024/a/The-Memory-Management-Unit/Translating-a-Virtual-Address-to-a-Physical-Address/Configuring-and-enabling-the-MMU` (visited on 04/19/2022).

[32] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. "A Security Framework for the Analysis and Design of Software Attestation". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS 13).* ACM, 2013, pp. 1–12. DOI: `10.1145/2508859.2516650`.

[33] N Asokan, Thomas Nyman, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik. "ASSURED: Architecture for Secure Software Update of Realistic Embedded Devices". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018), pp. 2290–2300. DOI: `10.1109/TCAD.2018.2858422`.

[34] AspenCore. *2019 Embedded Markets Study.* online. EETimes, 2019. URL: `https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf` (visited on 05/07/2021).

[35] Atmel Corporation. *ATmega328P 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash Datasheet.* 2015. URL: `https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf` (visited on 02/15/2022).

[36] Jia-Ju Bai, Tuo Li, Kangjie Lu, and Shi-Min Hu. "Static Detection of Unsafe DMA Accesses in Device Drivers". In: *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21).* USENIX Association, 2021, pp. 1629–1645. ISBN: 978-1-939133-24-3. URL: `https://www.usenix.org/conference/usenixsecurity21/presentation/bai`.

[37] Glynn Bartlett. *Extending the Industrial Robot Life Cycle.* 2021. URL: `https://www.swri.org/industry/industrial-robotics-automation/blog/extending-the-industrial-robot-life-cycle` (visited on 08/31/2022).

[38] Michael Becher, Maximillian Dornseif, and Christian N Klein. "FireWire: All Your Memory Are Belong to Us". In: *Proceedings of CanSecWest* 67 (2005).

[39] Ivan Cibrario Bertolotti. "Real-Time Embedded Operating Systems: Standards and Perspectives". In: *Embedded Systems Handbook* (2005). ISSN: 9781315217598.

[40] Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. "Address obfuscation: An efficient approach to combat a broad range of memory error exploits". In: *Proceedings of the 12th USENIX Security Symposium (USENIX Security 03).* USENIX Association, 2003. URL: `https://www.usenix.org/conference/12th-usenix-security-symposium/address-obfuscation-efficient-approach-combat-broad-range`.

[41] Bitcraze AB. *Datasheet Crazyflie 2.1 - Rev 3.* 2021. URL: `https://www.bitcraze.io/documentation/hardware/crazyflie_2_1/crazyflie_2_1-datasheet.pdf` (visited on 02/16/2022).

[42] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. "Jump-Oriented Programming: A New Class of Code-Reuse Attack". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS 11).* ACM, 2011, pp. 30–40. ISBN: 9781450305648. DOI: `10.1145/1966913.1966919`.

[43] Benjamin Böck. "Firewire-based Physical Security Attacks on Windows 7, EFS and BitLocker". In: *Secure Business Austria Research Lab* (2009). URL: `http://www.securityresearch.at/publications/windows7_firewire_physical_attacks.pdf`.

[44] Kaitlin Boeckl, Michael Fagan, William Fisher, Naomi Lefkovitz, Katerina Megas, Ellen Nadeau, Benjamin Piccarreta, Danna O'Rourke, and Karen Scarfone. *Considerations for Managing Internet of Things (IoT) Cybersecurity and Privacy Risks.* 2019. DOI: `10.6028/NIST.IR.8228`.

[45] Marton Bognar, Jo Van Bulck, and Frank Piessens. "Mind the gap: Studying the insecurity of provably secure embedded trusted execution architectures". In: *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE. 2022, pp. 1638–1655. DOI: `10.1109/SP46214.2022.9833735`.

[46] Marton Bognar, Hans Winderix, Jo Van Bulck, and Frank Piessens. "MicroProfiler: Principled Side-Channel Mitigation through Microarchitectural Profiling". In: *Proceedings of the 8th IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2023.

[47] Carsten Bormann, Mehmet Ersue, and Ari Keranen. "RFC 7228: Terminology for constrained-node networks". In: *Internet Engineering Task Force (IETF)* 7228 (2014). DOI: `10.17487/RFC7228`.

[48] Amani Braham, Félix Buendía, Maha Khemaja, and Faiez Gargouri. "User interface design patterns and ontology models for adaptive mobile applications". In: *Personal and Ubiquitous Computing* (2021), pp. 1318–1328. DOI: `10.1007/s00779-020-01481-5`.

[49] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. "TyTAN: Tiny Trust Anchor for Tiny Devices". In: *Proceedings of the 52nd Annual Design Automation Conference (DAC)*. ACM, 2015. DOI: `10.1145/2744769.2744922`.

[50] Ferdinand Brasser, Kasper Bonne Rasmussen, Ahmad-Reza Sadeghi, and Gene Tsudik. "Remote Attestation for Low-end Embedded Devices: the Prover's Perspective". In: *Proceedings of the 53rd Annual Design Automation Conference (DAC)*. ACM, 2016. DOI: `10.1145/2897937.2898083`.

[51] Business Wire. *Strategy Analytics: Global Smart Speaker Sales Cross 150 Million Units for 2020 Following Robust Q4 Demand*. 2021. URL: `https://www.businesswire.com/news/home/20210303005852/en/Strategy-Analytics-Global-Smart-Speaker-Sales-Cross-150-Million-Units-for-2020-Following-Robust-Q4-Demand` (visited on 07/16/2023).

[52] Kelly Caine. "Local Standards for Sample Size at CHI". In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 2016. DOI: `10.1145/2858036.2858498`.

[53] Todd Campau. *Average Age of Vehicles in the US Increases to 12.2 years, according to S&P Global Mobility*. 2022. URL: `https://ihsmarkit.com/research-analysis/average-age-of-vehicles-in-the-us-increases-to-122-years.html` (visited on 08/31/2022).

[54] Xavier Carpent, Norrathep Rattanavipanon, and Gene Tsudik. “Remote attestation of IoT devices via SMARM: Shuffled measurements against roving malware”. In: *Proceedings of the 2018 IEEE international symposium on hardware oriented security and trust (HOST)*. IEEE. 2018, pp. 9–16. DOI: 10.1109/HST.2018.8383885.

[55] Xavier Carpent, Gene Tsudik, and Norrathep Rattanavipanon. “ERASMUS: Efficient remote attestation via self-measurement for unattended settings”. In: *Proceedings of the 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2018, pp. 1191–1194. DOI: 10.23919/DATE.2018.8342195.

[56] Javier Carrillo-Mondéjar, Hannu Turtiainen, Andrei Costin, José Luis Martínez, and Guillermo Suarez-Tangil. “HALE-IoT: Hardening Legacy Internet of Things Devices by Retrofitting Defensive Firmware Modifications and Implants”. In: *IEEE Internet of Things Journal* (2022). DOI: 10.1109/JIOT.2022.3224649.

[57] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. “On the Difficulty of Software-based Attestation of Embedded Devices”. In: *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 09)*. ACM, 2009, pp. 400–409. DOI: 10.1145/1653662.1653711.

[58] Adam Caulfield, Norrathep Rattanavipanon, and Ivan De Oliveira Nunes. “ASAP: Reconciling Asynchronous Real-Time Operations and Proofs of Execution in Simple Embedded Systems”. In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. ACM, 2022, pp. 721–726. ISBN: 9781450391429. DOI: 10.1145/3489517.3530550.

[59] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. “Return-Oriented Programming without Returns”. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 10)*. ACM, 2010, pp. 559–572. ISBN: 9781450302456. DOI: 10.1145/1866307.1866370.

[60] Chien-Ying Chen, Monowar Hasan, and Sibin Mohan. “Securing Real-Time Internet-of-Things”. In: *Sensors* 18.12 (2018). DOI: 10.3390/s18124356.

[61] Guoxing Chen, Yinqian Zhang, and Ten-Hwang Lai. “OPERA: Open Remote Attestation for Intel’s Secure Enclaves”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS 19)*. ACM, 2019, pp. 2317–2331. DOI: 10.1145/3319535.3354220.

[62] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. "Live Updating Operating Systems Using Virtualization". In: *Proceedings of the International Conference on Virtual Execution Environments (VEE)*. ACM, 2006. DOI: `10.1145/1134760.1134767`.

[63] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. "POLUS: A POwerful Live Updating System". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE. 2007. DOI: `10.1109/ICSE.2007.65`.

[64] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. "Non-Control-Data Attacks Are Realistic Threats". In: *Proceedings of the 14th USENIX Security Symposium (USENIX 05)*. USENIX Association, 2005. URL: `https://www.usenix.org/conference/14th-usenix-security-symposium/non-control-data-attacks-are-realistic-threats`.

[65] Yaohui Chen, Yuping Li, Long Lu, Yueh-Hsun Lin, Hayawardh Vijayakumar, Zhi Wang, and Xinming Ou. "InstaGuard: Instantly Deployable Hot-patches for Vulnerable System Programs on Android". In: *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS)*. Internet Society. 2018. DOI: `10.14722/ndss.2018.23141`.

[66] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. "Adaptive Android Kernel Live Patching". In: *Proceedings of the 26th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, 2017, pp. 1253–1270. ISBN: 978-1-931971-40-9. URL: `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/chen`.

[67] Long Cheng, Christin Wilson, Song Liao, Jeffrey Young, Daniel Dong, and Hongxin Hu. "Dangerous Skills Got Certified: Measuring the Trustworthiness of Skill Certification in Voice Personal Assistant Platforms". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS 20)*. ACM, 2020. ISBN: 9781450370899. DOI: `10.1145/3372297.3423339`.

[68] Pascal Chevochot and Isabelle Puaut. "Scheduling Fault-Tolerant Distributed Hard Real-Time Tasks Independently of the Replication Strategies". In: *Proceedings of the International Conference on Real-Time Computing Systems and Applications (RTCSA)*. IEEE. 1999. DOI: `10.1109/RTCSA.1999.811280`.

[69] Boheung Chung, Jeongyeo Kim, and Youngsung Jeon. "On-demand security configuration for IoT devices". In: *Proceedings of the 2016 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE. 2016, pp. 1082–1084. DOI: `10.1109/ICTC.2016.7763373`.

[70] Abraham A. Clements, Naif Saleh Almakhdhub, Khaled S. Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. “Protecting Bare-Metal Embedded Systems with Privilege Overlays”. In: *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 289–303. DOI: `10.1109/SP.2017.37`.

[71] George Coker, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Brian O’Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen. “Principles of Remote Attestation”. In: *International Journal of Information Security* 10.2 (2011). DOI: `10.1007/s10207-011-0124-7`.

[72] Mauro Conti, Nicola Dragoni, and Viktor Lesyk. “A Survey of Man In The Middle Attacks”. In: *IEEE Communications Surveys & Tutorials* 18.3 (2016), pp. 2027–2051. DOI: `10.1109/COMST.2016.2548426`.

[73] Mauro Conti, Edlira Dushku, and Luigi V Mancini. “RADIS: Remote Attestation of Distributed IoT Services”. In: *Proceedings of the 2019 Sixth International Conference on Software Defined Systems (SDS)*. IEEE, 2019, pp. 25–32. DOI: `10.1109/SDS.2019.8768670`.

[74] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. “Inception: System-Wide Security Testing of Real-World Embedded Systems Software”. In: *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018, pp. 309–326. ISBN: 978-1-939133-04-5. URL: `https://www.usenix.org/conference/usenixsecurity18/presentation/corteggiani`.

[75] Victor Costan and Srinivas Devadas. “Intel SGX Explained”. In: *IACR Cryptology ePrint Archive, Paper 2016/086* (2016). URL: `http://eprint.iacr.org/2016/086`.

[76] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. “A Large-Scale Analysis of the Security of Embedded Firmwares”. In: *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, 2014, pp. 95–110. ISBN: 978-1-931971-15-7. URL: `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/costin`.

[77] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. “Protecting systems from stack smashing attacks with StackGuard”. In: *Linux Expo*. 1999.

[78] Cybersecurity and Infrastructure Security Agency. *Abbott Laboratories’ Accent/Anthem, Accent MRI, Assurity/Allure, and Assurity MRI Pacemaker Vulnerabilities*. 2017. URL: `https://www.cisa.gov/news-events/ics-medical-advisories/icsma-17-241-01` (visited on 03/15/2023).

[79] Sanjeev Das, Wei Zhang, and Yang Liu. “A Fine-Grained Control Flow Integrity Approach Against Runtime Memory Attacks for Embedded Systems”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.11 (2016), pp. 3193–3207. DOI: 10.1109/TVLSI.2016.2548561.

[80] Corbin Davenport. *This smartphone has physical kill switches for its cameras, microphone, data, Bluetooth, and Wi-Fi.* 2020. URL: https://www.androidpolice.com/2020/08/22/this-smartphone-has-physical-kill-switches-for-its-cameras-microphone-data-bluetooth-and-wi-fi/ (visited on 03/07/2022).

[81] Robert I. Davis and Alan Burns. “A survey of hard real-time scheduling for multiprocessor systems”. In: *ACM Computing Surveys* 43.4 (2011), 35:1–35:44. DOI: 10.1145/1978802.1978814.

[82] Dawoud Shenouda Dawoud and Peter Dawoud. *Serial Communication Protocols and Standards RS232/485, UART/USART, SPI, USB, INSTEON, Wi-Fi and WiMAX.* River Publishers, 2020.

[83] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Norrathep Rattanavipanon, and Gene Tsudik. “On the TOCTOU Problem in Remote Attestation”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS 21).* ACM, 2021, pp. 2921–2936. DOI: 10.1145/3460120.3484532.

[84] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. “Litehax: lightweight hardware-assisted attestation of program execution”. In: *Proceedings of the 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* IEEE. ACM, 2018. DOI: 10.1145/3240765.3240821.

[85] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N. Asokan, and Ahmad-Reza Sadeghi. “LO-FAT: Low-Overhead Control Flow ATtestation in Hardware”. In: *Proceedings of the 54th Annual Design Automation Conference (DAC).* ACM, 2017, 24:1–24:6. DOI: 10.1145/3061639.3062276.

[86] Maximillian Dornseif. “Owned by an ipod: Firewire/1394 issues”. In: 2005.

[87] Edlira Dushku, Md Masoom Rabbani, Mauro Conti, Luigi V Mancini, and Silvio Ranise. “SARA: Secure Asynchronous Remote Attestation for IoT Systems”. In: *IEEE Transactions on Information Forensics and Security* 15 (2020), pp. 3123–3136. DOI: 10.1109/TIFS.2020.2983282.

[88] Eclipse Foundation. *IoT & Edge Developer Survey Report.* 2021. URL: https://outreach.eclipse.foundation/iot-edge-developer-2021 (visited on 01/21/2023).

[89] Jide S Edu, Jose M Such, and Guillermo Suarez-Tangil. "Smart Home Personal Assistants: A Security and Privacy Review". In: *ACM Computing Surveys (CSUR)* 53.6 (2020). DOI: `10.1145/3412383`.

[90] Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. "HYDRA: hybrid design for remote attestation (using a formally verified microkernel)". In: *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 2017)*. ACM, 2017. DOI: `10.1145/3098243.3098261`.

[91] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. "SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust". In: *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2012. URL: `https://www.ndss-symposium.org/ndss2012/smart-secure-and-minimal-architecture-establishing-dynamic-root-trust`.

[92] ELM Technology. *GD25Q32 Datasheet*. 2014. URL: `https://datasheetspdf.com/pdf-file/861582/ELM/GD25Q32/1` (visited on 12/01/2022).

[93] ELM Technology. *GD25Q32C Datasheet*. 2020. URL: `http://www.elm-tech.com/en/products/spi-flash-memory/gd25q32/gd25q32.pdf` (visited on 12/01/2022).

[94] Mohammed Faisal Elrawy, Ali Ismail Awad, and Hesham F. A. Hamed. "Intrusion detection systems for IoT-based smart environments: a survey". In: *Journal of Cloud Computing* 7 (2018). DOI: `10.1186/s13677-018-0123-6`.

[95] Göran N Ericsson. "Cyber Security and Power System Communication—Essential Parts of a Smart Grid Infrastructure". In: *IEEE Transactions on Power Delivery* (2010), pp. 1501–1507. DOI: `10.1109/TPWRD.2010.2046654`.

[96] Espressif Systems. *Espressif Achieves the 100-Million Target for IoT Chip Shipments*. 2018. URL: `https://www.espressif.com/en/news/Espressif_Achieves_the_Hundredmillion_Target_for_IoT_Chip_Shipments` (visited on 05/07/2021).

[97] Espressif Systems. *ESP32 Technical Reference Manual*. 2020. URL: `https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf` (visited on 02/16/2022).

[98] Espressif Systems. *ESP8266 Technical Reference Manual*. 2020. URL: `https://www.espressif.com/sites/default/files/documentation/esp8266-technical_reference_en.pdf` (visited on 02/16/2022).

[99] Espressif Systems. *ESP8266EX Datasheet.* 2020. URL: https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf (visited on 07/23/2023).

[100] Espressif Systems. *Over The Air Updates (OTA).* 2020. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/ota.html (visited on 07/09/2020).

[101] Espressif Systems. *ESP-IDF Programming Guide: Memory Capabilities.* 2021. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/mem_alloc.html (visited on 07/06/2021).

[102] Espressif Systems. *ESP-IDF Programming Guide: Partition Tables.* 2021. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/partition-tables.html (visited on 05/06/2021).

[103] Espressif Systems. *ESP32 Series Datasheet.* 2021. URL: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf (visited on 05/03/2021).

[104] Espressif Systems. *ESP32-C3 Technical Reference Manual.* 2022. URL: https://www.espressif.com/sites/default/files/documentation/esp32-c3_technical_reference_manual_en.pdf (visited on 06/21/2022).

[105] Espressif Systems. *Wi-Fi Driver - ESP32 - ESP-IDF Programming Guide latest documentation.* 2022. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/wifi.html (visited on 03/16/2022).

[106] Nicolas Falliere, Liam O Murchu, and Eric Chien. “W32. Stuxnet Dossier”. In: *White paper, Symantec Security Response* 5.6 (2011).

[107] James P Farwell and Rafal Rohozinski. “Stuxnet and the future of cyber war”. In: *Survival* 53.1 (2011), pp. 23–40.

[108] Laura Faulkner. “Beyond the five-user assumption: Benefits of increased sample sizes in usability testing”. In: *Behavior Research Methods, Instruments, & Computers* 35.3 (2003), pp. 379–383. DOI: 10.3758/BF03195514.

[109] Meik Felser, Rüdiger Kapitza, Jürgen Kleinöder, and Wolfgang Schröder-Preikschat. “Dynamic Software Update of Resource-Constrained Distributed Embedded Systems”. In: *Embedded System Design: Topics, Techniques and Trends (IFIP TC10).* Springer, 2007. DOI: 10.1007/978-0-387-72258-0_33.

[110] Bo Feng, Alejandro Mera, and Long Lu. “P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling”. In: *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20).* USENIX Association, 2020, pp. 1237–1254. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/feng.

[111] Wei Feng, Yu Qin, Shijun Zhao, Ziwen Liu, Xiaobo Chu, and Dengguo Feng. "Secure Code Updates for Smart Embedded Devices based on PUFs". In: *Proceedings of the 16th International Conference on Cryptology and Network Security (CANS 2017)*. Springer. 2018, pp. 325–346.

[112] Aurélien Francillon, Quan Nguyen, Kasper B Rasmussen, and Gene Tsudik. "A minimalist approach to Remote Attestation". In: *Proceedings of the 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2014, pp. 1–6. DOI: `10.7873/DATE.2014.257`.

[113] Thomas Franke, Christiane Attig, and Daniel Wessel. "A Personal Resource for Technology Interaction: Development and Validation of the Affinity for Technology Interaction (ATI) Scale". In: *International Journal of Human–Computer Interaction* 35.6 (2019). DOI: `10.1080/10447318.2018.1456150`.

[114] FreeRTOS. *Over the Air (OTA) Updates*. 2020. URL: `https://www.freertos.org/ota/index.html` (visited on 07/09/2020).

[115] FreeRTOS. *API Reference*. 2022. URL: `https://www.freertos.org/a00106.html` (visited on 03/17/2022).

[116] FreeRTOS. *GitHub - FreeRTOS*. 2022. URL: `https://github.com/FreeRTOS/FreeRTOS/tree/master` (visited on 03/22/2022).

[117] Ulf Frisk. "Direct Memory Attack the Kernel". In: *Proceedings of DEFCON* 24 (2016).

[118] David Geer. "Industry Trends: Chip Makers Turn to Multicore Processors". In: *Computer* 38.5 (2005), pp. 11–13. DOI: `10.1109/MC.2005.160`.

[119] Gemalto. *The State of IoT Security*. 2018. URL: `https://www.infopoint-security.de/media/gemalto-state-of-iot-security-report.pdf` (visited on 02/16/2022).

[120] Thomas Gerace and Huseyin Cavusoglu. "The critical elements of the patch management process". In: *Communications of the ACM* 52.8 (2009), pp. 117–121. DOI: `10.1145/1536616.1536646`.

[121] Hamid Reza Ghaeini, Matthew Chan, Raad Bahmani, Ferdinand Brasser, Luis Garcia, Jianying Zhou, Ahmad-Reza Sadeghi, Nils Ole Tippenhauer, and Saman A Zonouz. "PAtt: Physics-based Attestation of Control Systems". In: *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. USENIX Association, 2019, pp. 165–180.

[122] Cristiano Giuffrida, Calin Iorgulescu, Giordano Tamburrelli, and Andrew S. Tanenbaum. "Automating Live Update for Generic Server Programs". In: *IEEE Transactions on Software Engineering* (2017). DOI: `10.1109/TSE.2016.2584066`.

[123] Cristiano Giuffrida and Andrew S Tanenbaum. “A Taxonomy of Live Updates”. In: *Proceedings of the 16th Annual Conference of the Advanced School for Computing and Imaging (ASCI)*. 2010.

[124] GNU Project - GNU Compiler Collection. *Specifying Attributes of Variables*. 2022. URL: `https://gcc.gnu.org/onlinedocs/gcc-11.3.0/gcc/Variable-Attributes.html#Variable-Attributes` (visited on 04/28/2022).

[125] Dan Goodin. *Brace yourselves—source code powering potent IoT DDoSes just went public*. 2016. URL: `http://arstechnica.com/security/2016/10/brace-yourselves-source-code-powering-potent-iot-ddoses-just-went-public/` (visited on 04/21/2022).

[126] Google. *Overview of the Play Integrity API*. 2022. URL: `https://developer.android.com/google/play/integrity/overview` (visited on 11/10/2022).

[127] Google. *SafetyNet Attestation API*. 2022. URL: `https://developer.android.com/training/safetynet/attestation` (visited on 11/10/2022).

[128] Tianxiao Gu, Chun Cao, Chang Xu, Xiaoxing Ma, Linghao Zhang, and Jian Lu. “Javelus: A Low Disruptive Approach to Dynamic Software Updates”. In: *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*. IEEE. 2012. DOI: `10.1109/APSEC.2012.55`.

[129] Zhonglei Gu and Yang Liu. “Scalable Group Audio-Based Authentication Scheme for IoT Devices”. In: *Proceedings of the 2016 12th International Conference on Computational Intelligence and Security (CIS)*. IEEE. 2016, pp. 277–281. DOI: `10.1109/CIS.2016.0070`.

[130] Rachid Guerraoui and André Schiper. “Software-Based Replication for Fault Tolerance”. In: *Computer* (1997). DOI: `10.1109/2.585156`.

[131] Zhixiu Guo, Zijin Lin, Pan Li, and Kai Chen. “SkillExplorer: Understanding the Behavior of Skills in Large Scale”. In: *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020, pp. 2649–2666. ISBN: 978-1-939133-17-5. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/guo`.

[132] Deepak Gupta, Pankaj Jalote, and Gautam Barua. “A formal framework for on-line software version change”. In: *IEEE Transactions on Software engineering* 22.2 (1996), pp. 120–131. DOI: `10.1109/32.485222`.

[133] Daniel Halperin, Thomas S. Heydt-Benjamin, Benjamin Ransford, Shane S. Clark, Benessa Defend, Will Morgan, Kevin Fu, Tadayoshi Kohno, and William H. Maisel. “Pacemakers and Implantable Cardiac Defibrillators: Software Radio Attacks and Zero-Power Defenses”. In: *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE. 2008, pp. 129–142. DOI: `10.1109/SP.2008.31`.

[134] Jun Han, Albert Jin Chung, Manal Kumar Sinha, Madhumitha Harishankar, Shijia Pan, Hae Young Noh, Pei Zhang, and Patrick Tague. “Do You Feel What I Hear? Enabling Autonomous IoT Device Pairing Using Different Sensor Types”. In: *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018. DOI: 10.1109/SP.2018.00041.

[135] Khalid Hasan, Kamanashis Biswas, Khandakar Ahmed, Nazmus S Nafi, and Md Saiful Islam. “A comprehensive review of wireless body area network”. In: *Journal of Network and Computer Applications* 143 (2019), pp. 178–198. DOI: 10.1016/j.jnca.2019.06.016.

[136] Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. “Kitsune: Efficient, General-Purpose Dynamic Software Updating for C”. In: *Proceedings of the 2012 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2012. DOI: 10.1145/2384616.2384635.

[137] Simon Holmbacka, Wictor Lund, Sébastien Lafond, and Johan Lilius. “Lightweight Framework for Runtime Updating of C-Based Software in Embedded Systems”. In: *Proceedings of the 5th Workshop on Hot Topics in Software Upgrades (HotSWUp 13)*. USENIX Association, 2013.

[138] Matthew B Hoy. “Alexa, Siri, Cortana, and More: An Introduction to Voice Assistants”. In: *Medical Reference Services Quarterly* 37.1 (2018), pp. 81–88. DOI: 10.1080/02763869.2018.1404391.

[139] Stefan Hristozov, Moritz Wettermann, and Manuel Huber. “A TOCTOU Attack on DICE Attestation”. In: *Proceedings of the 12th ACM Conference on Data and Application Security and Privacy (CODASPY 22)*. ACM, 2022, pp. 226–235. DOI: 10.1145/3508398.3511507.

[140] 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: *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2016, pp. 969–986. DOI: 10.1109/SP.2016.62.

[141] Wen Hu, Hailun Tan, Peter Corke, Wen Chan Shih, and Sanjay Jha. “Toward trusted wireless sensor networks”. In: *ACM Transactions on Sensor Networks (TOSN)* 7.1 (2010), pp. 1–25. ISSN: 1550-4859. DOI: 10.1145/1806895.1806900.

[142] James Huang. *NXP Microcontrollers Overview*. 2017. URL: https://www.nxp.com/docs/en/supporting-information/BL-Micro-NXP-Microcontroller-Overview-James-Huang.pdf (visited on 04/19/2022).

[143] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Gene Tsudik. "US-AID: Unattended Scalable Attestation of IoT Devices". In: *Proceedings of the 2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2018, pp. 21–30. DOI: 10.1109/SRDS.2018.00013.

[144] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Shaza Zeitouni. "SeED: Secure Non-Interactive Attestation for Embedded Devices". In: *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 17)*. 2017, pp. 64–74. ISBN: 9781450350846. DOI: 10.1145/3098243.3098260.

[145] Zineeddine Ould Imam, Marc Lacoste, and Ghada Arfaoui. "Towards a Modular Attestation Framework for Flexible Data Protection for Drone Systems". In: *Proceedings of the 17th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. IEEE, 2021, pp. 96–102. DOI: 10.1109/WiMob52687.2021.9606269.

[146] Infineon. *MPU_Memory_Protection for KIT_AURIX_TC297_TFT*. 2020. URL: https://www.infineon.com/dgdl/?fileId=5546d46274cf54d50174da37dc1d222e (visited on 04/19/2022).

[147] Infineon. *How to use direct memory access (DMA) controller in TRAVEO II family*. 2021. URL: https://www.infineon.com/dgdl/Infineon-AN220191_How_to_Use_Direct_Memory_Access_(DMA)_Controller_in_Traveo_II_Family-ApplicationNotes-v07_00-EN.pdf (visited on 04/19/2022).

[148] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. "Block Oriented Programming: Automating Data-Only Attacks". In: *Proceedings of the 2018 25th ACM SIGSAC Conference on Computer and Communications Security (CCS 18)*. ACM, 2018, pp. 1868–1882. ISBN: 9781450356930. DOI: 10.1145/3243734.3243739.

[149] ITU-T. *Overview of the Internet of things*. Recommendation Y.2060. International Telecommunication Union, 2012.

[150] Haegeon Jeong, Jeanseong Baik, and Kyungtae Kang. "Functional level hot-patching platform for executable and linkable format binaries". In: *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE. 2017. DOI: 10.1109/SMC.2017.8122653.

[151] Joy-IT. *Heartbeat Sensor KY-039*. 2018. URL: http://anleitung.joy-it.net/wp-content/uploads/2018/11/SEN-KY039-Manual.pdf (visited on 07/23/2020).

[152] JSOF-Tech. *Ripple20 - 19 Zero-Day Vulnerabilities Amplified by the Supply Chain*. 2020. URL: https://www.jsof-tech.com/ripple20/ (visited on 07/14/2020).

[153] Nikolaos Karapanos, Claudio Marforio, Claudio Soriente, and Srdjan Capkun. “Sound-proof: Usable two-factor authentication based on ambient sound”. In: *Proceedings of the 24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, 2015, pp. 483–498. URL: `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/karapanos`.

[154] Ori Karliner. *FreeRTOS TCP/IP Stack Vulnerabilities - The Details*. 2018. URL: `https://blog.zimperium.com/freertos-tcpip-stack-vulnerabilities-details/` (visited on 07/23/2020).

[155] Chongkyung Kil, Emre C Sezer, Ahmed M Azab, Peng Ning, and Xiaolan Zhang. “Remote attestation to dynamic system properties: Towards providing complete system integrity evidence”. In: *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE. 2009, pp. 115–124. DOI: `10.1109/DSN.2009.5270348`.

[156] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. “Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors”. In: *ACM SIGARCH Computer Architecture News* 42.3 (2014). DOI: `10.1145/2678373.2665726`.

[157] Bret Kinsella. *Alexa Skill Counts Surpass 80K in US, Spain Adds the Most Skills, New Skill Rate Falls Globally*. 2021. URL: `https://voicebot.ai/2021/01/14/alexa-skill-counts-surpass-80k-in-us-spain-adds-the-most-skills-new-skill-introduction-rate-continues-to-fall-across-countries/` (visited on 03/16/2022).

[158] Bret Kinsella. *Google Assistant Actions Grew Quickly in Several Languages in 2019, Matched Alexa Growth in English*. 2022. URL: `https://voicebot.ai/2020/01/19/google-assistant-actions-grew-quickly-in-several-languages-in-2019-match-alexa-growth-in-english/` (visited on 03/16/2022).

[159] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. “TrustLite: A Security Architecture for Tiny Embedded Devices”. In: *Proceedings of the 9th European Conference on Computer Systems (EuroSys 14)*. ACM, 2014. DOI: `10.1145/2592798.2592824`.

[160] Florian Kohnhäuser, Niklas Büscher, and Stefan Katzenbeisser. “A practical attestation protocol for autonomous embedded systems”. In: *Proceedings of the 2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2019, pp. 263–278.

[161] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak N. Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. “Experimental Security Analysis of a Modern Automobile”. In: *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE. 2010. DOI: `10.1109/SP.2010.34`.

[162] Daniel B Kramer and Kevin Fu. “Cybersecurity Concerns and Medical Devices: Lessons From a Pacemaker Advisory”. In: *JAMA* 318.21 (2017), pp. 2077–2078. ISSN: 0098-7484. DOI: `10.1001/jama.2017.15692`.

[163] Boyu Kuang, Anmin Fu, Shui Yu, Guomin Yang, Mang Su, and Yuqing Zhang. “ESDRA: An Efficient and Secure Distributed Remote Attestation Scheme for IoT Swarms”. In: *IEEE Internet of Things Journal* 6.5 (2019), pp. 8372–8383. DOI: `10.1109/JIOT.2019.2917223`.

[164] Boyu Kuang, Anmin Fu, Lu Zhou, Willy Susilo, and Yuqing Zhang. “DO-RA: Data-oriented runtime attestation for IoT devices”. In: *Computers & Security* 97 (2020), p. 101945. ISSN: 0167-4048. DOI: `10.1016/j.cose.2020.101945`.

[165] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. “NetCAT: Practical Cache Attacks from the Network”. In: *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 20–38. DOI: `10.1109/SP40000.2020.00082`.

[166] David Kushner. “The Making of Arduino”. In: *IEEE Spectrum* 26 (2011).

[167] Donghyun Kwon, Jangseop Shin, Giyeol Kim, Byoungyoung Lee, Yeongpil Cho, and Yunheung Paek. “uXOM: Efficient eXecute-Only Memory on ARM Cortex-M”. In: *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 2019, pp. 231–247. ISBN: 978-1-939133-06-9. URL: `https://www.usenix.org/conference/usenixsecurity19/presentation/kwon`.

[168] Jisu Kwon, Jeonghun Cho, and Daejin Park. “Function Block-Based Robust Firmware Update Technique for Additional Flash-Area/Energy-Consumption Overhead Reduction”. In: *Proceedings of the International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*. IEEE. 2019. DOI: `10.1109/ISPACS48206.2019.8986373`.

[169] Ralph Langner. “Stuxnet: Dissecting a Cyberwarfare Weapon”. In: *IEEE Security & Privacy* 9.3 (2011), pp. 49–51. DOI: `10.1109/MSP.2011.67`.

[170] Josephine Lau, Benjamin Zimmerman, and Florian Schaub. “Alexa, Are You Listening?: Privacy Perceptions, Concerns and Privacy-seeking Behaviors with Smart Speakers”. In: *Proceedings of the ACM on Human-Computer Interaction* 2.CSCW (2018). DOI: `10.1145/3274371`.

[171] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. “Keystone: An Open Framework for Architecting Trusted Execution Environments”. In: *Proceedings of the 15th European Conference on Computer Systems (EuroSys 20)*. ACM, 2020. DOI: `10.1145/3342195.3387532`.

[172] Frédéric Leens. “An introduction to I2C and SPI protocols”. In: *IEEE Instrumentation & Measurement Magazine* 12.1 (2009), pp. 8–13.

[173] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. “Multiprogramming a 64kB Computer Safely and Efficiently”. In: *Proceedings of the 26th Symposium on Operating Systems Principles (SOPS 17)*. ACM, 2017. DOI: `10.1145/3132747.3132786`.

[174] Fuchun J Lin, PM Chu, and Ming T Liu. “Protocol Verification Using Reachability Analysis: The State Space Explosion Problem and Relief Strategies”. In: *Proceedings of the ACM Workshop on Frontiers in Computer Communications Technology (SIGCOMM ’87)*. ACM, 1987, pp. 126–135. ISBN: 0897912454. DOI: `10.1145/55482.55496`.

[175] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. “Last-Level Cache Side-Channel Attacks are Practical”. In: *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE. 2015, pp. 605–622. DOI: `10.1109/SP.2015.43`.

[176] Kristis Makris and Rida A. Bazzi. “Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction”. In: *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 09)*. USENIX Association, 2009. URL: `https://www.usenix.org/conference/usenix-09/immediate-multi-threaded-dynamic-software-updates-using-stack-reconstruction`.

[177] Theo Markettos, Colin Rothwell, Brett F Gutstein, Allison Pearce, Peter G Neumann, Simon Moore, and Robert Watson. “Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals”. In: (2019).

[178] Peter Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. Springer Nature, 2021.

[179] Shijia Mei, Zhihong Liu, Yong Zeng, Lin Yang, and Jian Feng Ma. “Listen!: Audio-based Smart IoT Device Pairing Protocol”. In: *Proceedings of the 2019 IEEE 19th International Conference on Communication Technology (ICCT)*. IEEE. 2019, pp. 391–397. DOI: `10.1109/ICCT46805.2019.8947178`.

[180] Alejandro Mera, Yi Hui Chen, Ruimin Sun, Engin Kirda, and Long Lu. “D-Box: DMA-enabled Compartmentalization for Embedded Applications”. In: *Proceedings of the 2022 Network and Distributed Systems Security Symposium (NDSS)*. Internet Society. 2022.

[181] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. "DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis". In: *Proceedings of the 42nd IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021. DOI: `10.1109/SP40001.2021.00018`.

[182] Ralph Charles Merkle. *Secrecy, authentication, and public key systems.* Stanford University, 1979.

[183] Kathleen Metrick, Jared Semrau, and Shambavi Sadayappan. *Think Fast: Time Between Disclosure, Patch Release and Vulnerability Exploitation — Intelligence for Vulnerability Management, Part Two.* Mandiant, 2021. URL: `https://www.mandiant.com/resources/blog/time-between-disclosure-patch-release-and-vulnerability-exploitation` (visited on 04/05/2022).

[184] Microchip Technology Inc. *ATmega48A/PA/88A/PA/168A/PA/328/P.* 2018. URL: `https://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061A.pdf` (visited on 02/15/2022).

[185] Charlie Miller and Chris Valasek. *Remote Exploitation of an Unaltered Passenger Vehicle.* 2015. URL: `https://illmatics.com/Remote%20Car%20Hacking.pdf` (visited on 11/09/2022).

[186] Richard Mitev, Markus Miettinen, and Ahmad-Reza Sadeghi. "Alexa Lied to Me: Skill-based Man-in-the-Middle Attacks on Virtual Assistants". In: *Proceedings of the ACM Asia Conference on Computer and Communications Security (AsiaCCS).* ACM, 2019. DOI: `10.1145/3321705.3329842`.

[187] Motorola, Inc. "SPI Block Guide V03.06". In: *Document number S12SPIV3/D* (2003).

[188] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices." In: *Proceedings of the 2019 Network and Distributed Systems Security Symposium (NDSS).* Internet Society. 2018.

[189] Imanol Mugarza, Andoni Amurrio, Ekain Azketa, and Eduardo Jacob. "Dynamic Software Updates to Enhance Security and Privacy in High Availability Energy Management Applications in Smart Cities". In: *IEEE Access* (2019). DOI: `10.1109/ACCESS.2019.2905925`.

[190] Imanol Mugarza, Jorge Parra, and Eduardo Jacob. "Cetratus: A framework for zero downtime secure software updates in safety-critical systems". In: *Proceedings of the International Symposium on Industrial Embedded Systems (SIES).* IEEE. 2018. DOI: `10.1002/spe.2820`.

[191] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. "The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching". In: *Proceedings of the IEEE Symposium on Security and Privacy (SP).* IEEE. 2015. DOI: `10.1109/SP.2015.48`.

[192] National Institute of Standards and Technology. *Secure Hash Standard (FIPS 180-3).* 2008. URL: `https://csrc.nist.gov/publications/detail/fips/180/3/archive/2008-10-31` (visited on 09/14/2021).

[193] National Institute of Standards and Technology. *Secure Hash Standard (SHS).* 2015. URL: `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf` (visited on 09/12/2021).

[194] National Institute of Standards and Technology. *CVE-2018-16601.* 2018. URL: `https://nvd.nist.gov/vuln/detail/CVE-2018-16601` (visited on 03/29/2023).

[195] James Nesfield. "Sending Data Over Sound: How and Why?" In: *Electronic Design* (2019). URL: `https://www.electronicdesign.com/industrial-automation/article/21808186/sending-data-over-sound-how-and-why` (visited on 03/09/2022).

[196] Erik G Nilsson. "Design patterns for user interface for mobile applications". In: *Advances in engineering software* 40.12 (2009). DOI: `10.1016/j.advengsoft.2009.01.017`.

[197] NodeMCU Documentation. *WiFi Module.* 2022. URL: `https://nodemcu.readthedocs.io/en/release/modules/wifi/` (visited on 03/16/2022).

[198] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. "Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base". In: *Proceedings of the 22nd USENIX Security Symposium (USENIX Security 13).* USENIX Association, 2013, pp. 479–498. ISBN: 978-1-931971-03-4. URL: `https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/noorman`.

[199] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. "Sancus 2.0: A Low-Cost Security Architecture for IoT Devices". In: *ACM Transactions on Privacy and Security (TOPS)* 20.3 (2017), pp. 1–33. ISSN: 2471-2566. DOI: `10.1145/3079763`.

[200] Ivan De Oliveira Nunes, Ghada Dessouky, Ahmad Ibrahim, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik. "Towards Systematic Design of Collective Remote Attestation Protocols". In: *Proceedings of the 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2019, pp. 1188–1198. DOI: `10.1109/ICDCS.2019.00120`.

[201] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. "VRASED: A Verified Hardware/Software Co-Design for Remote Attestation". In: *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 2019, pp. 1429–1446. ISBN: 978-1-939133-06-9. URL: `https://www.usenix.org/conference/usenixsecurity19/presentation/de-oliveira-nunes`.

[202] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. "PURE: Using Verified Remote Attestation to Obtain Proofs of Update, Reset and Erasure in low-End Embedded Systems". In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. ACM, 2019, pp. 1–8. DOI: `10.1109/ICCAD45719.2019.8942118`.

[203] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. "APEX: A Verified Architecture for Proofs of Execution on Remote Devices under Full Software Compromise". In: *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020, pp. 771–788. ISBN: 978-1-939133-17-5. URL: `https://www.usenix.org/conference/senixsecurity20/presentation/nunes`.

[204] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. "DIALED: Data Integrity Attestation for Low-end Embedded Devices". In: *Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2021. DOI: `10.1109/DAC18074.2021.9586180`.

[205] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. "Tiny-CFA: Minimalistic Control-Flow Attestation Using Verified Proofs of Execution". In: *Proceedings of the 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2021. DOI: `10.23919/DATE51398.2021.9474029`.

[206] NXP. *Examples of Setting the DMA Controller on the Power Architecture MPC5675K Family of Microcontrollers*. 2012. URL: `https://www.nxp.com/docs/en/application-note/AN4522.pdf` (visited on 04/19/2022).

[207] Adam Osborne. *Introductions to Microcomputers: Volume One, Basic Concepts*. McGraw-Hill Osborne Media, 1980.

[208] OWASP. *Internet of Things (IoT) Top 10 2018*. 2018. URL: `https://owasp.org/www-pdf-archive/OWASP-IoT-Top-10-2018-final.pdf` (visited on 02/23/2022).

[209] OWASP. *About the OWASP Foundation.* 2023. URL: `https://owasp.org/about/` (visited on 02/01/2022).

[210] OWASP. *OWASP Top Ten.* 2023. URL: `https://owasp.org/www-project-top-ten/` (visited on 02/01/2023).

[211] Dorottya Papp, Zhendong Ma, and Levente Buttyán. "Embedded systems security: Threats, vulnerabilities, and attack taxonomy". In: *Proceedings of the Annual Conference on Privacy, Security and Trust (PST).* IEEE. 2015. DOI: `10.1109/PST.2015.7232966`.

[212] Mathias Payer, Boris Bluntschli, and Thomas R Gross. "DynSec: On-the-fly Code Rewriting and Repair". In: *Proceedings of the 5th Workshop on Hot Topics in Software Upgrades (HotSWUp 13).* USENIX Association, 2013.

[213] Mathias Payer and Thomas R. Gross. "Hot-patching a web server: A case study of ASAP code repair". In: *Proceedings of the Annual Conference on Privacy, Security and Trust (PST).* IEEE. 2013. DOI: `10.1109/PST.2013.6596048`.

[214] Wouter Penard and Tim van Werkhoven. "On the secure hash algorithm family". In: *Cryptography in context* (2008).

[215] Daniele Perito and Gene Tsudik. "Secure Code Update for Embedded Devices via Proofs of Secure Erasure". In: *Proceedings of the 15th European Symposium on Research in Computer Security (ESORICS).* Springer. 2010, pp. 643–662. ISBN: 978-3-642-15497-3.

[216] Pine Store ltd. *PinePhone.* 2022. URL: `https://pine64.com/product-category/pinephone/` (visited on 03/07/2022).

[217] Lumpapun Punchoojit and Nuttanont Hongwarittorrn. "Usability Studies on Mobile User Interface Design Patterns: A Systematic Literature Review". In: *Advances in Human-Computer Interaction* (2017). DOI: `10.1155/2017/6787504`.

[218] Davide Quarta, Marcello Pogliani, Mario Polino, Federico Maggi, Andrea Maria Zanchettin, and Stefano Zanero. "An Experimental Security Analysis of an Industrial Robot Controller". In: *Proceedings of the IEEE Symposium on Security and Privacy (SP).* IEEE. 2017. DOI: `10.1109/SP.2017.20`.

[219] Ashwin Ramaswamy, Sergey Bratus, Sean W. Smith, and Michael E. Locasto. "Katana: A Hot Patching Framework for ELF Executables". In: *Proceedings of the International Conference on Availability, Reliability and Security (ARES).* IEEE. 2010. DOI: `10.1109/ARES.2010.112`.

[220] Kyle Rankin. *Lockdown Mode on the Librem 5: Beyond Hardware Kill Switches.* 2019. URL: `https://puri.sm/posts/lockdown-mode-on-the-librem-5-beyond-hardware-kill-switches/` (visited on 03/07/2022).

[221] Edwin D. Reilly. “Memory-Mapped I/O”. In: *Encyclopedia of Computer Science*. John Wiley and Sons Ltd., 2003, p. 1152. ISBN: 0470864125.

[222] RISC-V. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*. 2017. URL: `https://riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf` (visited on 04/19/2022).

[223] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. “Return-Oriented Programming: Systems, Languages, and Applications”. In: *ACM Transactions on Information and System Security (TISSEC)* 15.1 (2012), pp. 1–34. ISSN: 1094-9224. DOI: `10.1145/2133375.2133377`.

[224] Roman Rogowski, Micah Morton, Forrest Li, Fabian Monrose, Kevin Z Snow, and Michalis Polychronakis. “Revisiting Browser Security in the Modern Era: New Data-Only Attacks and Defenses”. In: *Proceedings of the 2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2017, pp. 366–381. DOI: `10.1109/EuroSP.2017.39`.

[225] Florian Rommel, Lennart Glauer, Christian Dietrich, and Daniel Lohmann. “Wait-Free Code Patching of Multi-Threaded Processes”. In: *Proceedings of the Workshop on Programming Languages and Operating Systems, (SOSP)*. ACM, 2019. DOI: `10.1145/3365137.3365404`.

[226] Markus Rothmuller and Sam Barker. *IoT – the Internet of Transformation 2020*. Juniper Research Ltd, 2020. URL: `https://www.juniperresearch.com/whitepapers/iot-the-internet-of-transformation-2020` (visited on 02/23/2022).

[227] Björn Ruytenberg. *Breaking Thunderbolt Protocol Security: Vulnerability Report*. Eindhoven University of Technology. 2020. URL: `https://thunderspy.io/assets/reports/breaking-thunderbolt-security-bjorn-ruytenberg-20200417.pdf` (visited on 08/31/2022).

[228] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. “Trusted Execution Environment: What It is, and What It is Not”. In: *Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA*. Vol. 1. IEEE. 2015. DOI: `10.1109/Trustcom.2015.357`.

[229] Ahmad-Reza Sadeghi and Christian Stüble. “Property-Based Attestation for Computing Platforms: Caring about Properties, Not Mechanisms”. In: *Proceedings of the 2004 Workshop on New Security Paradigms (NSPW 04)*. ACM, 2004, pp. 67–77. ISBN: 1595930760. DOI: `10.1145/1065907.1066038`.

[230] Selma Saidi, Rolf Ernst, Sascha Uhrig, Henrik Theiling, and Benoît Dupont de Dinechin. "The Shift to Multicores in Real-Time and Safety-Critical Systems". In: *Proceedings of the 2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS).* IEEE, 2015. DOI: `10.1109/CODESISSS.2015.7331385`.

[231] Christopher Salls, Yan Shoshitaishvili, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. "Piston: Uncooperative Remote Runtime Patching". In: *Proceedings of the Annual Computer Security Applications Conference (ACSAC).* ACM, 2017. DOI: `10.1145/3134600.3134611`.

[232] Samsung. *Samsung Knox Developer Communication: Attestation.* 2021. URL: `https://docs.samsungknox.com/dev/knox-sdk/attestation.htm` (visited on 02/03/2023).

[233] M Angela Sasse and Ivan Flechais. "Usable security: Why do we need it? How do we get it?" In: O'Reilly, 2005.

[234] Nitesh Saxena, J-E Ekberg, Kari Kostiainen, and N Asokan. "Secure Device Pairing Based on a Visual Channel". In: *Proceedings of the 2006 IEEE Symposium on Security and Privacy (SP).* IEEE. 2006. DOI: `10.1109/SP.2006.35`.

[235] Martin Schrepp, Andreas Hinderks, and Jörg Thomaschewski. "Design and Evaluation of a Short Version of the User Experience Questionnaire (UEQ-S)". In: *International Journal of Interactive Multimedia and Artificial Intelligence* 4.6 (2017). DOI: `10.9781/ijimai.2017.09.001`.

[236] Dominik Schürmann and Stephan Sigg. "Secure Communication Based on Ambient Audio". In: *IEEE Transactions on Mobile Computing* 12.2 (2011). DOI: `10.1109/TMC.2011.271`.

[237] Moses D Schwartz, John Mulder, Jason Trent, and William D Atkins. "Control system devices: Architectures and supply channels overview". In: *Sandia Report SAND2010-5183, Sandia National Laboratories* (2010). DOI: `10.2172/993312`.

[238] Mark E. Segal and Ophir Frieder. "On-the-Fly Program Modification: Systems for a Dynamic Updating". In: *IEEE Software* (1993). DOI: `10.1109/52.199735`.

[239] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. "SCUBA: Secure Code Update By Attestation in Sensor Networks". In: *Proceedings of the 5th ACM Workshop on Wireless Security.* ACM, 2006, pp. 85–94. ISBN: 1595935576. DOI: `10.1145/1161289.1161306`.

[240] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. "Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems". In: *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP*. ACM, 2005, pp. 1–16. DOI: 10.1145/1095810.1095812.

[241] Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. "SWATT: SoftWare-based ATTestation for Embedded Devices". In: *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE. IEEE Computer Society, 2004, pp. 272–282. DOI: 10.1109/SECPRI.2004.1301329.

[242] Hovav Shacham. "The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86)". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 07)*. 2007, pp. 552–561. ISBN: 9781595937032. DOI: 10.1145/1315245.1315313.

[243] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X Liu. "A Large Scale Exploratory Analysis of Software Vulnerability Life Cycles". In: *Proceedings of the 34th International Conference on Software Engineering (ICSE 12)*. IEEE. 2012. DOI: 10.1109/ICSE.2012.6227141.

[244] Zach Shelby, Klaus Hartke, and Carsten Bormann. *RFC 7252: The constrained application protocol (CoAP)*. Tech. rep. 2014. DOI: 10.17487/RFC7252.

[245] Weidong Shi, Hsien-Hsin S. Lee, Laura Falk, and Mrinmoy Ghosh. "An Integrated Framework for Dependable and Revivable Architectures Using Multicore Processors". In: *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA 2006)*. IEEE, 2006. DOI: 10.1109/ISCA.2006.8.

[246] Devu Manikantan Shila, Penghe Geng, and Teems Lovett. "I can detect you: Using intrusion checkers to resist malicious firmware attacks". In: *Proceedings of the 2016 IEEE Symposium on Technologies for Homeland Security (HST)*. IEEE. 2016. DOI: 10.1109/THS.2016.7568958.

[247] Kang G. Shin and Parameswaran Ramanathan. "Real-Time Computing: A New Discipline of Computer Science and Engineering". In: *Proceedings of IEEE, Special Issue on Real-Time Systems*. IEEE, 1994. DOI: 10.1109/5.259423.

[248] Siglent. *Siglent SDS1104X-E 100MHz Four channel oscilloscope*. 2023. URL: https://www.siglent.eu/product/1139249/siglent-sds1104x-e-100mhz-four-channel-oscilloscope (visited on 04/03/2023).

[249] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. "PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary". In: *Proceedings of the 2019 Network and Distributed Systems Security Symposium (NDSS)*. Internet Society. 2019.

[250] Spectra Industrie-PC und Automation. *Embedded Configuration Manager (ECM)*. 2021. URL: `https://www.spectra.de/cms/splash/embedded-configuration-manager/` (visited on 07/13/2021).

[251] Embedded Staff. *Catching the Z-Wave*. 2006. URL: `https://www.embedded.com/catching-the-z-wave/` (visited on 04/28/2021).

[252] John A. Stankovic and Raj Rajkumar. "Real-Time Operating Systems". In: *Real Time Systems* 28.2-3 (2004). DOI: `10.1023/B:TIME.0000045319.20260.73`.

[253] Smiljanic Stasha. *An In-Depth View into Smart Home Statistics*. 2021. URL: `https://policyadvice.net/insurance/insights/smart-home-statistics/` (visited on 02/25/2022).

[254] Rodrigo Vieira Steiner and Emil Lupu. "Attestation in Wireless Sensor Networks: A Survey". In: *ACM Computing Surveys (CSUR)* 49.3 (2016). DOI: `10.1145/2988546`.

[255] Rodrigo Vieira Steiner and Emil Lupu. "Towards more practical software-based attestation". In: *Computer Networks* 149 (2019). DOI: `10.1016/j.comnet.2018.11.003`.

[256] Meghan Stewart and Angela Fleischmann. *Update release cycle for Windows clients*. Microsoft, 2023. URL: `https://learn.microsoft.com/en-us/windows/deployment/update/release-cycle` (visited on 03/28/2023).

[257] Patrick Stewin and Iurii Bystrov. "Understanding DMA Malware". In: *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 12)*. Springer. 2013, pp. 21–41. ISBN: 978-3-642-37300-8.

[258] STMicroelectronics. *Using the STM32F2, STM32F4 and STM32F7 Series DMA controller*. 2016. URL: `https://www.st.com/resource/en/application_note/dm00046011-using-the-stm32f2-stm32f4-and-stm32f7-series-dma-controller-stmicroelectronics.pdf` (visited on 04/19/2022).

[259] STMicroelectronics. *STM32F446xC/E Technical Reference Manual*. 2019. URL: `https://www.st.com/resource/en/data_brief/nucleo-f446re.pdf` (visited on 12/16/2020).

[260] STMicroelectronics. *Using the STM32F0/F1/F3/Gx/Lx Series DMA controller.* 2020. URL: https://www.st.com/resource/en/application_note/cd00160362-using-the-stm32f0f1f3gxlx-series-dma-controller-stmicroelectronics.pdf (visited on 04/19/2022).

[261] STMicroelectronics. *Managing memory protection unit in STM32 MCUs.* 2021. URL: https://www.st.com/resource/en/application_note/dm00272912-managing-memory-protection-unit-in-stm32-mcus-stmicroelectronics.pdf (visited on 04/19/2022).

[262] STMicroelectronics. *STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced Arm®-based 32-bit MCUs - Reference manual.* 2021. URL: https://www.st.com/resource/en/reference_manual/CD00171190-.pdf (visited on 04/20/2022).

[263] Keith Stouffer, Victoria Pillitteri, Suzanne Lightman, Marshall Abrams, and Adam Hahn. "Guide to Industrial Control Systems (ICS) Security". In: (2015). DOI: 10.6028/NIST.SP.800-82r2.

[264] Dan Su, Jiqiang Liu, Sencun Zhu, Xiaoyang Wang, and Wei Wang. ""Are you home alone?" "Yes" Disclosing Security and Privacy Vulnerabilities in Alexa Skills". In: *arXiv preprint arXiv:2010.10788* (2020).

[265] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. "OAT: Attesting Operation Integrity of Embedded Devices". In: *Proceedings of the IEEE Symposium on Security and Privacy (SP).* IEEE. 2020. DOI: 10.1109/SP40000.2020.00042.

[266] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal War in Memory". In: *Proceedings of the IEEE Symposium on Security and Privacy (SP).* IEEE. 2013. DOI: 10.1109/SP.2013.13.

[267] Hailun Tan, Wen Hu, and Sanjay Jha. "A TPM-Enabled Remote Attestation Protocol (TRAP) in Wireless Sensor Networks". In: *Proceedings of the 6th ACM Workshop on Performance Monitoring and Measurement of Heterogeneous Wireless and Wired Networks (PM2HW2N 11).* ACM, 2011, pp. 9–16. ISBN: 9781450309028. DOI: 10.1145/2069087.2069090.

[268] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "Throwhammer: Rowhammer Attacks over the Network and Defenses". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18).* 2018. ISBN: ISBN 978-1-939133-01-4.

[269] Tensilica, Inc. "Xtensa Instruction Set Architecture (ISA) Reference Manual". In: *RC-2010.1 Release* (2010).

[270] Texas Instruments Incorporated. *Direct Memory Access (DMA) Controller Module.* 2018. URL: `https://www.ti.com/lit/ug/slau395f/slau395f.pdf` (visited on 04/28/2022).

[271] The LLVM Compiler Infrastructure Project. *Attributes in Clang.* 2022. URL: `https://clang.llvm.org/docs/AttributeReference.html#variable-attributes` (visited on 04/28/2022).

[272] The MITRE Corporation. *Update Software, Mitigation M0951 – ICS | MITRE ATT&CK.* 2022. URL: `https://attack.mitre.org/versions/v12/mitigations/M0951/` (visited on 03/07/2023).

[273] Chin-Wei Tien, Tsung-Ta Tsai, Yi Chen, and Sy-Yen Kuo. "UFO - Hidden Backdoor Discovery and Security Verification in IoT Device Firmware". In: *Proceedings of the 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW).* IEEE. 2018, pp. 18–23. DOI: `10.1109/ISSREW.2018.00-37`.

[274] Sven Türpe, Andreas Poller, Jan Steffan, Jan-Peter Stotz, and Jan Trukenmüller. "Attacking the BitLocker Boot Process". In: *Proceedings of the Second International Conference on Trusted Computing (Trust 2009).* Springer. 2009, pp. 183–196.

[275] US Food and Drug Administration. *Firmware Update to Address Cybersecurity Vulnerabilities Identified in Abbott's (formerly St. Jude Medical's) Implantable Cardiac Pacemakers: FDA Safety Communication.* 2017. URL: `https://www.fda.gov/medical-devices/safety-communications/firmware-update-address-cybersecurity-vulnerabilities-identified-abbotts-formerly-st-jude-medicals` (visited on 06/25/2020).

[276] Antti Valmari. "The state explosion problem". In: *Advanced Course on Petri Nets.* Springer. 1996.

[277] Victor Van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. "Practical context-sensitive CFI". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS 15).* ACM, 2015. DOI: `10.1145/2810103.2813673`.

[278] Michael Wahler and Manuel Oriol. "Disruption-free software updates in automation systems". In: *Proceedings of the IEEE Emerging Technology and Factory Automation (ETFA).* IEEE. 2014. DOI: `10.1109/ETFA.2014.7005075`.

[279] Michael Wahler, Stefan Richter, Sumit Kumar, and Manuel Oriol. "Non-disruptive large-scale component updates for real-time controllers". In: *Proceedings of the Workshops of the IEEE International Conference on Data Engineering (ICDE).* IEEE. 2011. DOI: `10.1109/ICDEW.2011.5767631`.

[280] Michael Wahler, Stefan Richter, and Manuel Oriol. “Dynamic Software Updates for Real-Time Systems”. In: *Proceedings of the Workshop on Hot Topics in Software Upgrades (HotSWUp).* ACM, 2009. DOI: 10.1145/1656437.1656440.

[281] Arielle Waldman. *Mitre AT&CK: How it has evolved and grown.* 2020. URL: https://www.techtarget.com/searchsecurity/news/252491169/Mitre-ATTCK-How-it-has-evolved-and-grown (visited on 03/15/2023).

[282] Jinwen Wang, Ao Li, Haoran Li, Chenyang Lu, and Ning Zhang. “RT-TEE: Real-time System Availability for Cyber-physical Systems using ARM TrustZone”. In: *Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP).* IEEE. 2022, pp. 352–369. DOI: 10.1109/SP46214.2022.9833604.

[283] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. “From Hack to Elaborate Technique - A Survey on Binary Rewriting”. In: *ACM Computing Surveys (CSUR)* 52.3 (2019). DOI: 10.1145/3316415.

[284] Jos Wetzels. *The RTOS Exploit Mitigation Blues.* 2017. URL: https://hardwear.io/document/rtos-exploit-mitigation-blues-hardwear-io.pdf (visited on 08/31/2022).

[285] Bas Wijnen, Emily J Hunt, Gerald C Anzalone, and Joshua M Pearce. “Open-Source Syringe Pump Library”. In: *PloS one* 9.9 (2014). DOI: 10.1371/journal.pone.0107216.

[286] Reinhard Wilhelm and Jan Reineke. “Embedded systems: Many cores - Many problems”. In: *Proceedings of the 7th IEEE International Symposium on Industrial Embedded Systems (SIES 2012).* IEEE, 2012. DOI: 10.1109/SIES.2012.6356583.

[287] Edgar Wolf and Francis C Marino. *Acoustic coupler.* US Patent 3,553,374. 1971.

[288] Minhua Wu, Sankaran Panchapagesan, Ming Sun, Jiacheng Gu, Ryan Thomas, Shiv Naga Prasad Vitaladevuni, Bjorn Hoffmeister, and Arindam Mandal. “Monophone-Based Background Modeling for Two-Stage On-Device Wake Word Detection”. In: *Proceedings of the 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* IEEE. 2018. DOI: 10.1109/ICASSP.2018.8462227.

[289] Weitao Xu, Chitra Javali, Girish Revadigar, Chengwen Luo, Neil Bergmann, and Wen Hu. “Gait-Key: A Gait-Based Shared Secret Key Generation Protocol for Wearable Devices”. In: *ACM Transactions on Sensor Networks (TOSN)* 13.1 (2017). DOI: 10.1145/3023954.

[290] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. "Automatic Hot Patch Generation for Android Kernels". In: *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020, pp. 2397–2414. ISBN: 978-1-939133-17-5. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/xu`.

[291] Yi Yang, Xinran Wang, Sencun Zhu, and Guohong Cao. "Distributed Software-based Attestation for Node Compromise Detection in Sensor Networks". In: *Proceedings of the 26th IEEE Symposium on Reliable Distributed Systems (SRDS 2007)*. IEEE Computer Society, 2007. DOI: `10.1109/SRDS.2007.31`.

[292] Joseph Yiu. *The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors*. Elsevier Science, 2013. ISBN: 9780124079182.

[293] Joseph Yiu. *ARM Cortex-M for Beginners*. 2016. URL: `https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-2142-00-00-00-00-52-96/White-Paper-_2D00_-Cortex_2D00_M-for-Beginners-_2D00_-2016-_2800_final-v3_2900_.pdf` (visited on 01/21/2023).

[294] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Mihai Christodorescu, and Lui Sha. "Learning Execution Contexts from System Call Distribution for Anomaly Detection in Smart Embedded System". In: *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*. ACM, 2017. DOI: `10.1145/3054977.3054999`.

[295] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Jung-Eun Kim, and Lui Sha. "SecureCore: A Multicore-based Intrusion Detection Architecture for Real-Time Embedded Systems". In: *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013*. IEEE Computer Society, 2013. DOI: `10.1109/RTAS.2013.6531076`.

[296] Man-Ki Yoon, Lui Sha, Sibin Mohan, and Jaesik Choi. "Memory Heat Map: Anomaly Detection in Real-Time Embedded Systems Using Memory Behavior". In: *Proceedings of the 52nd Annual Design Automation Conference (DAC)*. ACM, 2015. DOI: `10.1145/2744769.2744869`.

[297] Bruno Bogaz Zarpelão, Rodrigo Sanches Miani, Cláudio Toshio Kawakani, and Sean Carlisto de Alvarenga. "A survey of intrusion detection in Internet of Things". In: *Journal of Network and Computer Applications* 84 (2017). DOI: `10.1016/j.jnca.2017.02.009`.

[298] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. “ATRIUM: Runtime Attestation Resilient Under Memory Attacks”. In: *Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2017*. IEEE, 2017. DOI: `10.1109/ICCAD.2017.8203803`.

[299] Eric Zeng, Shrirang Mare, and Franziska Roesner. “End User Security and Privacy Concerns with Smart Homes”. In: *Proceedings of the 13th Symposium on Usable Privacy and Security (SOUPS 2017)*. 2017.

[300] Matthias Zeppelzauer, Alexis Ringot, and Florian Taurer. *SoniTalk – an open ultrasonic communication protocol*. 2019. URL: `https://sonitalk.fhstp.ac.at/` (visited on 03/07/2022).

[301] Matthias Zeppelzauer, Alexis Ringot, and Florian Taurer. *SoniTalk*. 2022. URL: `https://github.com/fhstp/SoniTalk` (visited on 03/07/2022).

[302] Kim Zetter. *A Cyberattack Has Caused Confirmed Physical Damage for the Second Time Ever*. 2015. URL: `https://www.wired.com/2015/01/german-steel-mill-hack-destruction/` (visited on 04/09/2021).

[303] Chi Zhang, Wonsun Ahn, Youtao Zhang, and Bruce R. Childers. “Live Code Update for IoT Devices in Energy Harvesting Environments”. In: *Proceedings of the Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE. 2016. DOI: `10.1109/NVMSA.2016.7547182`.

[304] Nan Zhang, Xianghang Mi, Xuan Feng, XiaoFeng Wang, Yuan Tian, and Feng Qian. “Dangerous Skills: Understanding and Mitigating Security Risks of Voice-Controlled Third-Party Functions on Virtual Personal Assistant Systems”. In: *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 1381–1396. DOI: `10.1109/SP.2019.00016`.

[305] Shaohu Zhang and Anupam Das. “HandLock: Enabling 2-FA for Smart Home Voice Assistants using Inaudible Acoustic Signal”. In: *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. ACM, 2021. ISBN: 9781450390583. DOI: `10.1145/3471621.3471866`.

[306] Serena Zheng, Noah Apthorpe, Marshini Chetty, and Nick Feamster. “User Perceptions of Smart Home IoT Privacy”. In: *Proceedings of the ACM on Human-Computer Interaction* 2, Issue CSCW (2018). DOI: `10.1145/3274469`.

Eidesstattliche Erklärung

Ich gebe folgende eidesstattliche Erklärung nach §14 Abs. 1 Nr. 6 der Promotionsordnung 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. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Essen, den 14. März 2024

Sebastian Erasmus Raphael Josef Surminski