ADVANCED REMOTE ATTESTATION PROTOCOLS FOR EMBEDDED SYSTEMS

Vom Fachbereich Informatik der
Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)
von

M. Sc. Florian Kohnhäuser
geboren in Groß-Gerau, Deutschland

Referenten:	Prof. Dr.-Techn. Stefan Katzenbeisser
	Universität Passau
	Prof. Dr.-Ing. Matthias Hollick
	Technische Universität Darmstadt
Tag der Einreichung:	06.06.2019
Tag der mündl. Prüfung:	18.07.2019

D 17
Darmstadt, 2019

ERKLÄRUNG

Hiermit versichere ich, die vorliegende Arbeit ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, 06. Juni 2019

Florian Kohnhäuser

AKADEMISCHER WERDEGANG

Seit 02/2015	**Technische Universität Darmstadt** Wissenschaftlicher Mitarbeiter Promotionsstudium
10/2012 - 11/2014	**Technische Universität Darmstadt** Studium der IT-Sicherheit Abschluss: Master of Science
10/2009 - 09/2012	**Karlsruher Institut für Technologie** Studium der Informatik Abschluss: Bachelor of Science

ABSTRACT

Small integrated computers, so-called embedded systems, have become a ubiquitous and indispensable part of our lives. Every day, we interact with a multitude of embedded systems. They are, for instance, integrated in home appliances, cars, planes, medical devices, or industrial systems. In many of these applications, embedded systems process privacy-sensitive data or perform safety-critical operations. Therefore, it is of high importance to ensure their secure and safe operation. However, recent attacks and security evaluations have shown that embedded systems frequently lack security and can often be compromised and misused with little effort. A promising technique to face the increasing amount of attacks on embedded systems is remote attestation. It enables a third party to verify the integrity of a remote device. Using remote attestation, attacks can be effectively detected, which allows to quickly respond to them and thus minimize potential damage. Today, almost all servers, desktop PCs, and notebooks have the required hardware and software to perform remote attestation. By contrast, a secure and efficient attestation of embedded systems is considerably harder to achieve, as embedded systems have to encounter several additional challenges.

In this thesis, we tackle three main challenges in the attestation of embedded systems. First, we address the issue that low-end embedded devices typically lack the required hardware to perform a secure remote attestation. We present an attestation protocol that requires only minimal secure hardware, which makes our protocol applicable to many existing low-end embedded devices while providing high security guarantees. We demonstrate the practicality of our protocol in two applications, namely, verifying code updates in mesh networks and ensuring the safety and security of embedded systems in road vehicles. Second, we target the efficient attestation of multiple embedded devices that are connected in challenging network conditions. Previous attestation protocols are inefficient or even inapplicable when devices are mobile or lack continuous connectivity. We propose an attestation protocol that particularly targets the efficient attestation of many devices in highly dynamic and disruptive networks. Third, we consider a more powerful adversary who is able to physically tamper with the hardware of embedded systems. Existing attestation protocols that address physical attacks suffer from limited scalability and robustness. We present two protocols that are capable of verifying the software integrity as well as the hardware integrity of embedded devices in an efficient and robust way. Whereas the first protocol is optimized towards scalability, the second protocol aims at robustness and is additionally suited to be applied in autonomous networks.

In summary, this thesis contributes to enhancing the security, efficiency, robustness, and applicability of remote attestation for embedded systems.

KURZFASSUNG

Kleine integrierte Computer, sogenannte eingebettete Systeme, sind zu einem allgegenwärtigen und unverzichtbaren Bestandteil unseres Lebens geworden. Jeden Tag interagieren wir mit einer Vielzahl von eingebetteten Systemen, die z.B. in Haushaltsgeräten, Autos, Flugzeugen, medizinischen Geräten oder industriellen Systemen integriert sind. In vielen dieser Anwendungen verarbeiten eingebettete Systeme datenschutzrelevante Informationen oder steuern sicherheitskritische Prozesse, sodass es wichtig ist, die IT-Sicherheit dieser eingebetteten Systeme zu gewährleisten. Jüngste Angriffe und Sicherheitsanalysen haben jedoch gezeigt, dass viele eingebettete Systeme ein niedriges IT-Sicherheitsniveau aufweisen und oft mit geringem Aufwand kompromittiert und zweckentfremdet werden können. Eine vielversprechende Technik, um der zunehmenden Bedrohung durch Angriffe auf eingebettete Systeme zu begegnen, sind Attestierungsverfahren (engl. Remote Attestation). Diese ermöglichen es einer dritten Partei, die Integrität eines entfernten Gerätes zu überprüfen. Mittels Attestierung können Angriffe effektiv erkannt werden, wodurch schnell auf Angriffe reagiert und potenzielle Schäden minimiert werden können. Die Attestierung von eingebetteten Systemen bringt jedoch eine Reihe von Herausforderungen mit sich, die in existierenden Arbeiten bisher nicht oder nur unzureichend behandelt wurden.

In dieser Arbeit stellen wir uns drei zentralen Herausforderungen. Zunächst befassen wir uns mit dem Aspekt, dass kostengünstige eingebettete Systeme in der Regel nicht über die erforderliche Hardware verfügen, um eine sichere Attestierung durchzuführen. Wir stellen ein Attestierungsprotokoll vor, das hohe Sicherheitsgarantien bietet und dabei nur ein Minimum an sicherer Hardware benötigt. Durch die niedrigen Hardwareanforderungen ist unser Protokoll auf vielen kostengünstigen eingebetteten Systemen einsetzbar. Wir demonstrieren die Praktikabilität unseres Protokolls in zwei Anwendungen, der Verifikation von Software Updates in Sensornetzen und der Gewährleistung funktionaler Sicherheit von Steuergeräten in Fahrzeugen. Als Nächstes betrachten wir die effiziente Attestierung mehrerer eingebetteter Systeme in großen Netzwerken. Bisherige Attestierungsprotokolle sind ineffizient oder nicht anwendbar, wenn keine dauerhafte Konnektivität zwischen allen Geräten im Netzwerk besteht oder Geräte ihre Position ändern. Wir präsentieren ein Attestierungsprotokoll, das eine effiziente Attestierung vieler Geräte ermöglicht, die in hochdynamischen und unterbrochenen Netzwerken miteinander verbunden sind. Als Letztes befassen wir uns mit dem Schutz vor invasiven physikalischen Angriffen auf die Hardware eingebetteter Systeme. Bestehende Attestierungsprotokolle, die physische Angriffe erkennen, besitzen lediglich eine eingeschränkte Skalierbarkeit und Robustheit. Wir stellen zwei Protokolle vor, die in der Lage sind, die Software- und Hardware-Integrität eingebetteter Geräte auf effiziente und robuste Weise zu

überprüfen. Während unser erstes Protokoll besonders effizient und skalierbar ist, bietet unser zweites Protokoll eine hohe Robustheit und eignet sich damit insbesondere für den Einsatz in autonomen Netzwerken.

Zusammenfassend verbessert diese Arbeit die Sicherheit, Effizienz, Robustheit und Anwendbarkeit von Attestierungsverfahren für eingebettete Systeme.

LIST OF PUBLICATIONS

PEER-REVIEWED PUBLICATIONS USED IN THIS THESIS

[83] Florian Kohnhäuser, Niklas Büscher, and Stefan Katzenbeisser. "SALAD: Secure and Lightweight Attestation of Highly Dynamic and Disruptive Networks." In: *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*. 2018.

[84] Florian Kohnhäuser, Niklas Büscher, and Stefan Katzenbeisser. "A Practical Attestation Protocol for Autonomous Embedded Systems." In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. 2019.

[85] Florian Kohnhäuser and Stefan Katzenbeisser. "Secure Code Updates for Mesh Networked Commodity Low-End Embedded Devices." In: *European Symposium on Research in Computer Security (ESORICS)*. 2016.

[86] Florian Kohnhäuser, Dominik Püllen, and Stefan Katzenbeisser. "Ensuring the Safe and Secure Operation of Electronic Control Units in Road Vehicles." In: *IEEE Security and Privacy Workshops (SPW)*. 2019.

[88] Florian Kohnhäuser, Niklas Büscher, Sebastian Gabmeyer, and Stefan Katzenbeisser. "SCAPI: A Scalable Attestation Protocol to Detect Software and Physical Attacks." In: *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WISEC)*. 2017.

[127] Steffen Schulz, André Schaller, Florian Kohnhäuser, and Stefan Katzenbeisser. "Boot Attestation: Secure Remote Reporting with Off-The-Shelf IoT Sensors." In: *European Symposium on Research in Computer Security (ESORICS)*. 2017.

FURTHER PEER-REVIEWED PUBLICATIONS

[47] Arved Eßer, Florian Kohnhäuser, Nadine Ostern, Kevin Engleson, and Stephan Rinderknecht. "Enabling a Privacy-Preserving Synthesis of Representative Driving Cycles from Fleet Data using Data Aggregation." In: *IEEE Intelligent Transportation Systems Conference (ITSC)*. 2018.

[87] Florian Kohnhäuser, André Schaller, and Stefan Katzenbeisser. "PUF-based software protection for low-end embedded devices." In: *International Conference on Trust and Trustworthy Computing (TRUST)*. 2015.

[89] Florian Kohnhäuser*, Milan Stute*, Lars Baumgärtner, Lars Almon, Stefan Katzenbeisser, Matthias Hollick, and Bernd Freisleben. "SEDCOS: A Secure Device-to-Device Communication System for Disaster Scenarios." In: *IEEE Local Computer Networks Conference (LCN)*. 2017. *Both authors contributed equally to this work.

[105] Christian Meurisch, Julien Gedeon, Artur Gogel, The An Binh Nguyen, Florian Kohnhäuser, Lars Baumgärtner, Milan Schmittner, and Max Mühlhäuser. "Temporal coverage analysis of router-based cloudlets using human mobility patterns." In: *IEEE Global Communications Conference (GLOBECOM)*. 2017.

ACKNOWLEDGMENTS

During my journey as a PhD student, I was fortunate to be inspired and supported by many people. Without them, this thesis would not have been possible. First and foremost, I am deeply grateful to my supervisor Stefan Katzenbeisser. His invaluable guidance, insightful comments, offered freedom, and confidence in my work contributed greatly to my research and this thesis. Second, I thank my co-supervisor Matthias Hollick for reviewing this thesis and for heading the LOEWE research cluster NICER, which provided me with a pleasant and stimulating research environment. Furthermore, I thank Thomas Schneider, Max Mühlhäuser, and Reiner Hähnle for joining my committee.

In addition, I am thankful to all my current and former colleagues. In particular, many thanks go to my colleagues from the Security Engineering Group, namely, André, Christian, Dominik, Erik, Heike, Marius, Markus, Nikolaos A., Nikolaos K., Niklas, Nikolay, Philipp, Sebastian B., Sebastian G., Spyros, Tolga, and Ursula. Thanks for all the help and support as well as the fun and laughter we had at work, barbecues, lunches, coffee breaks, ice breaks, and our retreats in Kleinwalsertal. Moreover, I thank the ENCRYPTO Group and the NICER research team for many enjoyable hours at lunches, barcamps, conferences, workshops, and social events.

Finally, I want to express my appreciation to my family and my friends who have always encouraged and helped me throughout these years. Above all, I am most thankful for my wife Viktoria and her endless support and patience.

CONTENTS

ACRONYMS

COTS	Commercial Off-The-Shelf
DDoS	Distributed Denial of Service
DoS	Denial of Service
DTN	Delay-Tolerant Network
ECU	Electronic Control Unit
HMAC	Hash-based Message Authentication Code
IETF	Internet Engineering Task Force
IoT	Internet of Things
MAC	Message Authentication Code
MANET	Mobile Ad hoc Network
MPU	Memory Protection Unit
OS	Operating System
ROM	Read-Only Memory
ROP	Return Oriented Programming
RTC	Real-Time Clock
SGX	Software Guard Extensions
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TCPA	Trusted Computing Platform Alliance
TEE	Trusted Execution Environment
TPM	Trusted Platform Module

1 INTRODUCTION

In the past years, the continuous cost reduction and miniaturization of electronic devices started a new technological era of omnipresent *embedded systems*. Embedded systems are specialized computers that are typically integrated in electrical and mechanical systems. As opposed to general-purpose computers, embedded systems are specifically engineered to serve a dedicated function. For this reason, they are usually optimized towards low costs, small size, low power consumption, and harsh operating conditions. Although embedded systems are already present in many aspects of our lives, e.g., in wearables, home appliances, vehicles, avionics, medical devices, or industrial control, their amount and their applications are expected to further increase, facilitated by numerous initiatives and trends like the Internet of Things (IoT), Smarter Planet, Industry 4.0, or Smart Cities. In fact, Gartner [54] estimates that more than 20 billion connected embedded systems, also referred to as IoT devices, will be deployed by 2020.

In many applications, embedded systems perform safety-critical operations or process privacy sensitive data, for instance, as they control the mechanics of road vehicles or function as intelligent personal assistants. In these cases, embedded systems offer a high potential for misuse, which makes establishing their security an essential objective. However, effective security solutions are much harder to realize on embedded systems than on general-purpose computers. This is because embedded systems commonly possess constrained computing resources as well as a small and simple system architecture, e.g., preventing the implementation of secure hardware and cryptographic accelerators. Furthermore, the embedded systems industry tends to suffer from cost pressure, short development cycles, and lack of security standards, which provokes a poor product quality. In fact, various studies have revealed numerous security vulnerabilities in embedded systems [34, 36, 59, 115]. Thus, it is not surprising that embedded systems have become appealing targets for real-world attacks. In recent years, unpatched and misconfigured routers, cameras, and other consumer devices were increasingly infected with malware that enables attackers to perform Distributed Denial of Service (DDoS) attacks, mine cryptocurrencies, or exfiltrate personal data [76]. In addition to these simple and widespread attacks, embedded systems were also subject to highly sophisticated and targeted attacks like Stuxnet, Industroyer, or Triton [61]. Stuxnet, for instance, exploited previously unknown vulnerabilities to make embedded systems damage uranium enriching centrifuges. These attacks highlight the need for effective security solutions for embedded systems.

Comprehensive security solutions typically cover three main aspects: prevention, detection, and response [94]. Preventive techniques, such as encryption, authentication, software sandboxing, or exploit mitigation, proactively harden

systems against attacks, and thus constitute the first line of defense. However, preventive techniques cannot defend against all attacks, meaning that they only raise the bar for successful attacks. Therefore, detection and response techniques also play a vital role [55]. They ensure that attacks are quickly identified and that appropriate reactions minimize and contain potential damage. For instance, in case of Stuxnet, detection techniques could have enabled operators to immediately notice that the uranium enrichment centrifuges were compromised. By quickly reacting and shutting down the centrifuges, the operators could have prevented their destruction. A key technique that enables the detection of compromised embedded devices is *remote attestation*.

1.1 REMOTE ATTESTATION

Remote attestation is a security service that enables a third party, the *verifier*, to check whether a remote device, the *prover*, is in a trustworthy system state. Traditionally, remote attestation is implemented as a challenge-response protocol that is executed between a single verifier and a single prover. Initially, the verifier sends a unique challenge to the prover. The prover measures its local software integrity, e.g., by computing a hash value over its installed software, and then authenticates the measurements and the challenge from the verifier with a unique attestation key. Afterwards, the prover sends its attestation response, which consists of the software measurement and the authentication value, to the verifier. A priori, the verifier stores the attestation key and expected (known-good) integrity measurement of the prover. By computing the expected attestation response based on the stored key and integrity measurement, and comparing it with the received response, the verifier finally determines whether the prover is in a trustworthy state (or not).

Remote attestation has initially been developed for general-purpose computers. More than a decade ago, the Trusted Computing Group (TCG), a standardization consortium of various technology companies, introduced the Trusted Platform Module (TPM) and the associated concept of remote attestation. The TPM is a dedicated chip that is integrated into the motherboard of a platform and enables a variety of trusted computing features, such as, sealing, binding, and remote attestation [139]. To enable these features, the TPM provides a secure storage for software measurements and cryptographic keys, and a cryptographic processor for digital signatures, encryption, and hash functions. Nowadays, TPMs are implemented in almost all servers, desktop PCs, and notebooks, and recent Windows or Linux operating systems support their capabilities.

However, the embedded systems market is different. Compared with general-purpose computers, embedded devices face various additional challenges that hamper a secure and efficient attestation. For instance, due to size and costs restrictions, additional secure hardware, such as a TPM, is commonly not deployed in embedded systems. Further obstacles include limited computing power, low memory, low communication bandwidth, challenging network topologies, and

device heterogeneity. Although some of these obstacles have been overcome by the numerous attestation protocols that were recently proposed [3, 7, 11, 28, 29, 38, 44, 51, 65, 67, 68, 71, 114, 129, 132, 150], many other challenges are still insufficiently addressed by existing works.

1.2 GOAL AND SCOPE OF THIS THESIS

The goal of this thesis is to advance the practicality of remote attestation on embedded systems, with the aim to facilitate its deployment and use in practice. For this purpose, we address three main questions, which constitute the scope of this thesis.

Q1: Can attestation protocols provide strong security guarantees on commodity low-end embedded systems?

Attestation protocols can be divided in two classes: *software-based* and *hardware-based* attestation protocols. Software-based protocols are independent of secure hardware, but on the downside rely on assumptions that have shown to be hard to achieve in practice [10, 29]. In contrast, hardware-based attestation protocols provide much stronger security guarantees by relying on secure hardware, such as a TPM, ARM TrustZone, or Intel SGX. Embedded systems are commonly optimized for minimal size and production costs. For this reason, commodity embedded devices, especially low-end and legacy devices, typically lack the secure hardware that is required by hardware-based protocols. Therefore, attestation protocols that are applicable to commodity low-end embedded devices are commonly software-based, and thus provide questionable security guarantees. In this thesis, we target the attestation of commodity low-end embedded devices with strong security guarantees. Furthermore, we explore novel applications for protocols that achieve a strong attestation of low-end embedded devices.

Q2: Can protocols achieve an efficient and robust attestation of many embedded systems that are connected in challenging network topologies?

Wireless communication technologies like Bluetooth Smart, IEEE 802.15.4, ZigBee, or Z-Wave enable embedded devices to form large, autonomous, and mobile networks. Applying traditional single-device attestation protocols in these infrastructure-less networks results in a huge overhead, as the verifier needs to perform the protocol with each device individually. For this reason, *collective* attestation protocols were proposed [7, 11, 28, 65, 67, 68]. They distribute the attestation burden across the entire network to achieve an efficient attestation of all network devices. Collective attestation protocols typically assume a fully connected and quasi static network topology. However, in many use cases, embedded devices are mobile and/or operate in sparsely populated networks. Existing attestation protocols are inefficient or even inapplicable in these networks. In this thesis, we aim at an efficient attestation of multiple devices that are connected in highly dynamic and disruptive network topologies.

Furthermore, embedded devices may operate autonomously, i.e., with minimal or no supervision, for long times. Due to lack of manual intervention, attestation protocols for autonomous embedded systems must be particularly robust and be able to sustain their security service in case of network and device failures. This prevents the use of a central entity that initiates the attestation and verifies its result. Instead, multiple embedded devices must be capable of mutually attesting and verifying the system state of each other. This is challenging given the fact that the verifier is typically a powerful device and embedded devices have limited resources. In this thesis, we address the specific requirements for an efficient and robust attestation of autonomous embedded systems.

Q3: Can practical attestation protocols defend against a physical adversary who is able to tamper with the hardware of embedded systems?

In many applications, embedded systems are publicly accessible, left unattended, and deployed in large quantities. These circumstances allow an adversary to physically approach and tamper with the hardware of embedded systems more easily than with general-purpose computers. Originally, physical attacks were outside the threat model of attestation protocols. Hence, by physically tampering with some devices, an adversary is often able to corrupt the entire attestation result. To solve this issue, collective attestation protocols have recently been combined with *absence detection* to detect both software and physical attacks [65, 68]. Absence detection builds on the assumption that an adversary who physically tampers with a device must temporarily take the device offline for a noticeable amount of time, e.g., to disassemble the device and extract secret keys. Attestation protocols that detect physical attacks require devices to periodically engage in the protocol execution. Devices that do not execute the protocol for longer than a certain threshold are regarded as absent, and thus physically compromised. However, existing protocols suffer from limited scalability and are prone to network and device outages, whereupon healthy, but temporarily unreachable, devices are mistakenly regarded as physically compromised. This renders existing protocols impractical in many applications. In this thesis, we investigate practical attestation protocols that defend against physical attacks while being scalable and robust.

1.3 SUMMARY OF CONTRIBUTIONS

In this thesis, we extend and improve the current state of the art in the attestation of embedded systems by addressing the aforementioned research questions (Q1-Q3). More specifically, we provide the following contributions:

Chapter 3: Attestation of Commodity Low-End Embedded Systems (Q1). We present an attestation protocol that is specifically tailored to low-cost and resource-constrained embedded systems. Our protocol requires only minimal secure hardware features, which are available in many existing low-end embedded devices. This enables our protocol to combine the advantages of hardware-based

and software-based attestation protocols. In particular, our protocol provides the same (strong) security guarantees as other hardware-based attestation protocols while being applicable to commodity low-end embedded devices, like software-based protocols. We implement our protocol on an exemplary low-end embedded device and demonstrate its practicality and performance. Furthermore, we provide an overview of typical commodity low-end embedded devices and explain steps to implement our protocol on them.

This chapter is based on the following publications:

[85] Florian Kohnhäuser and Stefan Katzenbeisser. "Secure Code Updates for Mesh Networked Commodity Low-End Embedded Devices." In: *European Symposium on Research in Computer Security (ESORICS)*. 2016.

[86] Florian Kohnhäuser, Dominik Püllen, and Stefan Katzenbeisser. "Ensuring the Safe and Secure Operation of Electronic Control Units in Road Vehicles." In: *IEEE Security and Privacy Workshops (SPW)*. 2019.

[127] Steffen Schulz, André Schaller, Florian Kohnhäuser, and Stefan Katzenbeisser. "Boot Attestation: Secure Remote Reporting with Off-The-Shelf IoT Sensors." In: *European Symposium on Research in Computer Security (ESORICS)*. 2017.

Chapter 4: Attestation Applications with Embedded Systems (Q1). We demonstrate the practicality of our proposed attestation protocol for low-end embedded systems (Chapter 3) by using it in two novel applications. In both applications, previous attestation protocols were inapplicable due to their weak security guarantees or their requirements on secure hardware. In the first use case, we apply our protocol in road vehicles to ensure the vehicles' secure and safe operation. Our solution verifies the software integrity of all safety-critical embedded devices in the vehicle before the vehicle is started. In case the verification of a device fails, the vehicle is prevented from moving. We show that many embedded systems in vehicles conform to automotive standards that mandate the necessary secure hardware required to apply our proposed attestation protocol. Finally, we implement our solution on an exemplary automotive network and demonstrate that it imposes an imperceptible overhead for drivers and passengers. In the second use case, we combine our proposed attestation protocol with existing code update techniques to achieve a secure code update solution for mesh networks. Our solution enforces the proper installation of code updates on all devices in the network. Compromised devices can either refuse an appropriate execution of the code update, whereupon they are excluded from the network, or properly install the update, whereby any present malware is eliminated. This way, our solution is able to recover compromised devices and reestablish trust in them. Furthermore, its strong security guarantees allow our solution to be applicable in scenarios where an adversary is able to compromise more than one network de-

vice. We demonstrate the efficiency and scalability of our solution by real-world measurements and network simulations.

This chapter is based on the following publications:

[85] Florian Kohnhäuser and Stefan Katzenbeisser. "Secure Code Updates for Mesh Networked Commodity Low-End Embedded Devices." In: *European Symposium on Research in Computer Security (ESORICS)*. 2016.

[86] Florian Kohnhäuser, Dominik Püllen, and Stefan Katzenbeisser. "Ensuring the Safe and Secure Operation of Electronic Control Units in Road Vehicles." In: *IEEE Security and Privacy Workshops (SPW)*. 2019.

Chapter 5: Attestation of Highly Dynamic and Disruptive Networks (Q2). We propose an attestation protocol for highly dynamic and disruptive networks. Our protocol uses a novel distributed approach, where all devices incrementally establish a common view on the integrity of all devices in the network. In contrast to existing protocols, this approach allows our protocol to perform well in highly dynamic and disruptive network topologies, to provide an increased resilience against targeted Denial of Service (DoS) attacks, and to enable the verifier the reception of the attestation result from any device. Moreover, our protocol mitigates physical attacks by preventing a physically compromised device from forging a valid system state for other devices in the network (than itself). We evaluate the protocol through network simulations, which are based on measurements of ZigBee connected low-end embedded devices. Our simulation results demonstrate the efficiency and performance of our protocol, even in networks where hundreds of embedded devices move with high (drone) speed in large areas.

This chapter is based on the following publication:

[83] Florian Kohnhäuser, Niklas Büscher, and Stefan Katzenbeisser. "SALAD: Secure and Lightweight Attestation of Highly Dynamic and Disruptive Networks." In: *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*. 2018.

Chapter 6: Scalable Attestation of Software and Physical Attacks (Q3). We present a scalable attestation protocol that detects software and physical attacks. Based on the assumption that physical attacks require an adversary to take a targeted device offline for a noticeable amount of time, our protocol identifies devices with compromised hardware. Compared to a previously proposed attestation protocol that detects physical attacks [68], our solution provides the following improvements: (i) it is very efficient, as it reduces the number of transmitted messages per time period from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$, where n denotes the total

number of devices in the network; (ii) it is robust against network delays by relying on a unidirectional 1-to-n delay-tolerant link in each session period, in contrast to an n-to-n continuous link; and (iii) it can precisely identify devices whose hardware and/or software is compromised, if less than half of all devices in the network are compromised. Network simulations show that our protocol is applicable to low-end embedded devices, can scale to millions of devices, and can outperform existing solutions by orders of magnitude.

This chapter is based on the following publication:

[88] Florian Kohnhäuser, Niklas Büscher, Sebastian Gabmeyer, and Stefan Katzenbeisser. "SCAPI: A Scalable Attestation Protocol to Detect Software and Physical Attacks." In: *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WISEC)*. 2017.

Chapter 7: Practical Attestation of Autonomous Embedded Systems (Q2 and Q3). We propose an attestation protocol that is particularly suited for embedded systems in autonomous networks. Our protocol is the first that (i) enables many embedded prover devices to attest their integrity towards many potentially untrustworthy embedded verifier devices, (ii) is fully decentralized, thus, able to withstand network disruptions and arbitrary device outages, and (iii) is in addition to software attacks capable of detecting physical attacks in a much more robust way than any existing protocol. We implement the protocol, conduct measurements on commodity devices, and simulate large networks based on the measurements. Our results demonstrate that the protocol is practical on low-end to mid-range embedded devices, scales to large networks with millions of devices, and improves robustness by multiple orders of magnitude compared with the best existing protocols.

This chapter is based on the following publication:

[84] Florian Kohnhäuser, Niklas Büscher, and Stefan Katzenbeisser. "A Practical Attestation Protocol for Autonomous Embedded Systems." In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. 2019.

1.4 THESIS OUTLINE

In Chapter 2, we introduce the necessary background for this thesis, where we describe related work, explain our system and adversary model, and state our security assumptions. In Chapters 3 to 7, we present our contributions, which have been summarized in the previous section. In Chapter 8, we conclude this thesis and outline directions for future research.

2 PRELIMINARIES

In this chapter, we first summarize related works and introduce basic concepts on which this thesis is built (Section 2.1). Next, we explain our system model (Section 2.2) and state the hardware requirements of our proposed attestation solutions (Section 2.3). Finally, we present our adversary model, which defines the capabilities of two adversaries, a software and a physical adversary (Section 2.4).

2.1 BACKGROUND

2.1.1 *Remote Attestation*

Overview. Remote attestation is a security service that enables a third party, the *verifier*, to verify the software integrity of a remote device, the *prover*. The concept of remote attestation was introduced almost two decades ago by the Trusted Computing Platform Alliance (TCPA) [5], a consortium of various technology companies with the goal to implement trusted computing concepts in personal computers. Later on, the TCPA was succeeded by the Trusted Computing Group (TCG), which is still the main driver of trusted computing technology today.

Remote attestation is typically implemented as an interactive protocol that is executed between verifier and prover. Figure 2.1 illustrates a simplified attestation protocol. As shown, the verifier initially prepares an attestation challenge, which consists of a nonce, and sends it to the prover. The prover measures its local software integrity by computing a cryptographic hash value over its entire program memory. Next, the prover generates its attestation response, which consists of a Message Authentication Code (MAC) that is computed over the nonce from the verifier and its local software integrity measurement with a unique attestation key. Afterwards, the prover sends its attestation response to the verifier, who verifies its correctness. For this purpose, the verifier stores the attestation key as well as the known-good software integrity measurement of the prover. This enables the verifier to recompute the expected attestation response of the prover. In case the recomputed attestation response matches the received attestation response, the verifier considers the prover to be in a trustworthy software state. In all other cases, the verifier regards the prover as untrustworthy.

Attestation protocols typically provide security against an adversary who is able to perform network attacks and/or compromise the software of the prover. Under these assumptions, attestation protocols prevent the adversary from forging a valid attestation response for a compromised prover. To achieve this, provers rely on a root of trust that is either implemented in software or hardware. For instance, in our exemplary attestation protocol, the root of trust must ensure

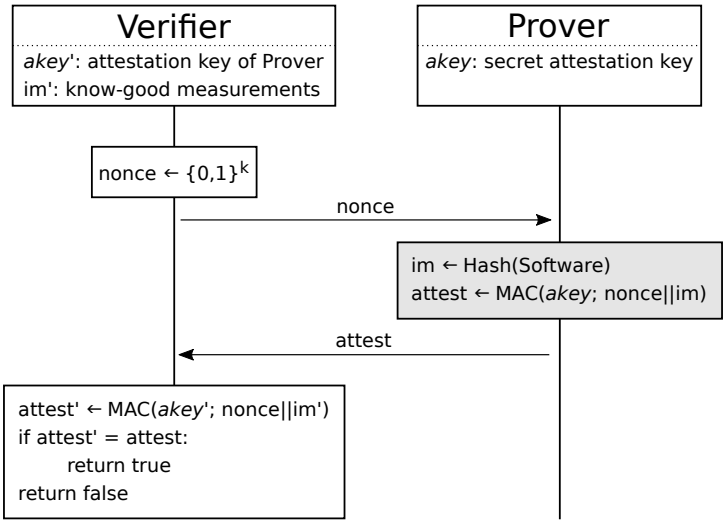


Figure 2.1: Illustration of a simplified (hardware-based) attestation protocol. Operations highlighted by the grey box are vital for security, which is why secure hardware must ensure their untampered execution. In addition, secure hardware must prevent malicious code from obtaining the secret attestation key *akey*.

that an adversary is unable to obtain the prover's secret attestation key. In addition, it must prevent the adversary from tampering with the computation of the integrity measurement and the generation of the attestation response (see Figure 2.1). In the following, we describe both types to ensure a secure attestation, i.e., *software-based* and *hardware-based* attestation, in detail.

Software-based Attestation. Attestation protocols that require no secure hardware are referred to as software-based attestation protocols [10, 98, 129–131]. To achieve security, software-based protocols not only depend on cryptographic operations or checksums, but also on time measurements. Their approach is at first sight similar to our illustrated attestation protocol (Figure 2.1): The prover receives a challenge from the verifier, computes a checksum over the program memory and the challenge, and sends the result to the verifier, who eventually verifies its correctness. However, in software-based attestation protocols, the verifier also measures the time between sending out the challenge and receiving the response from the prover. In case the time measurement is larger than a specific threshold, the prover is regarded to be in a compromised software state. The idea behind this assumption is that malicious code needs to interfere with the checksum computation in order to evade detection. This interference delays the checksum computation, which is eventually noticed by the verifier.

The advantage of software-based attestation is its broad applicability. Especially low-end and legacy embedded devices may not provide secure hardware. With software-based techniques, their integrity can still be verified. On the downside, the security of software-based attestation protocols relies on several strong assumptions. First, the prover must implement and execute the attestation protocol in an optimal way, such that malicious code is unable to compute the attestation

response faster than the protocol code. Second, the adversary must be passive during protocol execution, meaning that the adversary must be unable to compute the attestation response with a device other than the actual prover. Third, the protocol execution must not be subject to variable delays, e.g., introduced by a communication over several hops. In practice, these assumptions have shown to be hard to achieve [10, 29]. For this reason, software-based attestation is typically considered to only be applicable in specific scenarios, e.g., the attestation of peripheral devices.

Hardware-based Attestation. In contrast to software-based attestation protocols, hardware-based protocols [8, 91, 124, 139] require provers to be equipped with secure hardware. Popular secure hardware are the Trusted Platform Module (TPM), ARM TrustZone, and Intel Software Guard Extensions (SGX), which are nowadays (almost) built into all commercial servers, desktops, and mid-range and high-end embedded devices. Their security features prevent malicious software from accessing the secret attestation key and ensure the untampered execution of security critical parts of the attestation protocol (see Figure 2.1). In this way, hardware-based protocols are able to provide much stronger security guarantees than software-based protocols.

For low-end embedded devices, commodity secure hardware components are often too complex and expensive [51]. To address this issue, *hybrid* attestation schemes have been proposed, e.g., SMART [45], SANCUS [109], TrustLite [82], and TyTan [23]. They rely on a software/hardware co-design to enable a secure attestation with minimal requirements on secure hardware. Nevertheless, these hardware architectures have only been implemented as prototypes and their future availability in commodity low-end embedded devices is uncertain.

Runtime Attestation. Attestation protocols build on different techniques to measure the software integrity of provers. In general, existing protocols can be classified in two groups: *load-time* and *runtime* attestation protocols. Load-time protocols [124, 139], which are also referred to as static or boot-time protocols, measure the integrity of the software before it is executed. To this end, load-time protocols typically compute a cryptographic hash value over the software binary that is to be executed. Most attestation protocols used today are load-time protocols, including protocols based on the widely-deployed TPM [139].

By contrast, runtime, or dynamic, attestation protocols [3, 38, 39, 150] measure the integrity of the software during its execution. For this purpose, they capture the control flow of the software at runtime and report it to the verifier. This enables the verifier to trace the prover's execution path and to determine whether the execution path has been compromised. Thus, runtime attestation protocols are able to detect sophisticated attacks, in which the adversary hijacks the control flow of a program, e.g., using Return Oriented Programming (ROP). Control flow hijacking attacks cannot be detected with load-time attestation protocols, since these attacks have no effect on the software binary. On the downside, runtime attestation protocols are much more inefficient than load-time protocols and hence suffer from scalability issues in large programs.

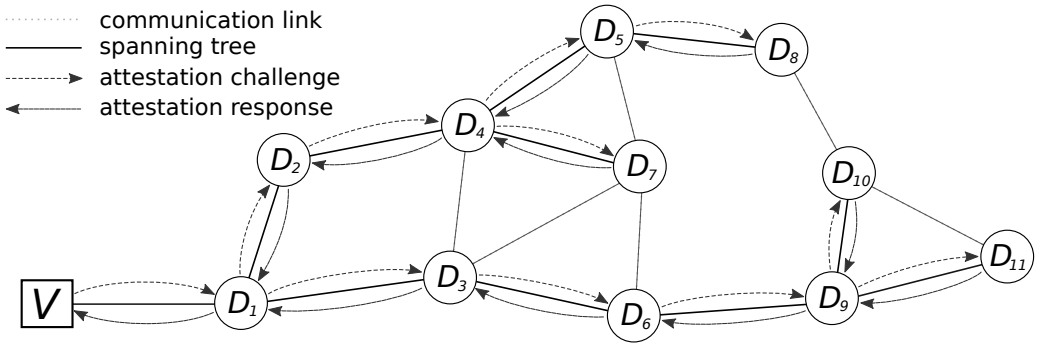


Figure 2.2: Illustration of a collective attestation protocol. A spanning tree is arrange in the mesh network, which enables the efficient transmission and aggregation of attestation responses from prover devices ($\mathcal{D}_1, ..., \mathcal{D}_{11}$) to the verifier ($\mathcal{V}$).

2.1.2 *Collective Attestation*

Scalable Attestation. In many applications, embedded systems are deployed in large quantities, e.g., in industrial control or automotive systems. In these cases, the verifier is typically interested in verifying multiple, or even all, devices in the network. Traditional attestation protocols (Section 2.1.1) focus on the attestation of a single device. Hence, in case they are used to verify multiple devices, the verifier needs to run the protocol with each device individually. This entails a high communication and runtime overhead, which renders single-device protocols impractical to verify large networks. Especially in networks where devices route data on behalf of other devices in the network, e.g., Mobile Ad hoc Networks (MANETs), the overhead is particularly large.

Collective attestation protocols address this issue by providing a scalable attestation of many devices. To this end, collective protocols distribute the attestation burden across all devices in the network, as illustrated in Figure 2.2 and described in the following. Initially, the verifier sends an attestation challenge to an arbitrary device, which is then propagated in the network from device to device. Propagating the challenge arranges a spanning tree overlay in the network, where a receiving device regards the sending device as its parent in the tree. Thus, the verifier is the root of the tree, and devices whose neighbors have all received the challenge are the leafs of the tree. Next, each leaf device generates its attestation response and sends it to its parent device. Parent devices collect the attestation responses of all their child devices, aggregate them with their own response, and send the aggregated response to their own parent device. In this way, attestation responses are aggregated and propagated hop-by-hop along the spanning tree to the verifier. Finally, the verifier obtains a single aggregated response to which all devices have contributed. With the aggregated response, the verifier can determine the software integrity of all network devices.

The described approach was first proposed in SEDA [11] and has been enhanced by subsequent collective attestation protocols in various ways. SANA [7]

allows multiple entities to verify attestation reports, enables devices without secure hardware to aggregate attestation reports, and limits the strength of physical attacks by authenticating individual attestation reports. LISA [28] provides a quality metric for collective attestation protocols and discusses protocol design decisions to achieve different quality features. SeED [67] introduces a non-interactive attestation approach that mitigates Denial of Service (DoS) attacks and increases efficiency, as the attestation is initiated by a secure clock on prover devices (instead of the verifier). DIAT [4] constitutes the first collective runtime attestation protocol and particularly focuses on the attestation of data integrity, i.e., the proper generation and processing of data in collaborative networks.

Physical Adversary. Although in certain scenarios, devices are physically unreachable or protected by a secure perimeter, in many applications, the physical security of devices cannot be assured. However, most attestation protocols only focus on software attacks and consider physical attacks to be out of scope. Thus, an adversary who is able to physically approach and tamper with devices can bypass their security goals. In fact, only two attestation protocols, DARPA [68] and US-AID [65], also defend against physical attacks. For this purpose, they combine collective attestation with absence detection to detect both, software and physical attacks. Absence detection builds on the assumption that an adversary requires to take a device offline for a continuous amount of time to successfully tamper with it [32, 33]. Hence, devices that are offline for longer than a specific threshold are considered to be physically compromised.

In DARPA [68], each device periodically emits an authenticated heartbeat that is logged by all other devices in the network. Since physically attacked device are offline for longer than the threshold time, they will miss sending a heartbeat in at least one period, which is noticed by all other devices. During attestation, the verifier receives the heartbeat log of all devices and considers devices that missed sending a heartbeat in at least one period as physically compromised. A weakness of DARPA is its limited scalability, as the amount of periodically exchanged messages scales quadratically with the number of network devices. US-AID [65] improves the scalability of DARPA by requiring devices to prove their physical presence only to neighboring devices, but not to all devices in the network. To support network dynamics, devices in US-AID periodically emit a token that attests the presence of all their neighboring devices. Neighboring devices record these tokens and use them to prove their presence towards newly encountered devices upon network topology changes.

2.1.3 *Secure Code Updates*

Secure code update techniques specifically address the problem of verifying that code updates are securely distributed and correctly installed on remote devices. A device that properly executed a secure code update has proven to be in a trustworthy, i.e., an unmodified and up-to-date, software state, clean of any malware. SCUBA [132] verifies the proper execution of the software update

protocol on a remote device using software-based attestation. Executing SCUBA, the verifier obtains a report which indicates that either (i) the code update was correctly installed on the remote device, such that any malicious code is wiped out, or (ii) malicious code interfered with the code update, e.g., to prevent its erasure, in which case the remote device is in a potentially untrustworthy software state. Nevertheless, software-based attestation, hence also SCUBA, provides questionable security guarantees due to its strong assumptions. Other secure code update techniques build on the concept of Proofs of Secure Erasure (PoSE) [75, 116]. PoSE enables a device to prove to a remote party that it has erased all its memory and thus is free of malicious code. In a second step, cleaned devices download the software update and send a MAC of the downloaded code to the verifier in order to prove the storage of the software update. The initial concept of PoSE [116] was enhanced by combining PoSE with All or Nothing Transforms to reduce the time and energy overhead [75]. Nevertheless, both software and PoSE-based approaches rely on the strong assumption that the prover is only able to communicate with the verifier, and with no other party. Thus, both approaches are unsuited for updating devices that are part of a larger network, since they can only provide security if the adversary is physically absent and has not gained control of more than one device in the network. By contrast, ASSURED [12] is a software update framework that provides stronger security guarantees by relying on hybrid attestation techniques. ASSURED extends existing update distribution schemes and is specifically designed for large-scale Internet of Things (IoT) settings with resource-constrained embedded devices. HEALED [66] constitutes another hardware-based secure code update solution and focuses on networks with multiple devices. In HEALED, devices in the network verify the software integrity of each other based on a Merkle Hash Tree (MHT) construction. The MHT allows to determine the exact software block that is defective on a prover. In case a compromised software block is detected, a benign healer device uses MHT information to deliver a minimal patch to the corrupt prover, and thereby restores its software.

2.1.4 *Secure Aggregation*

To reduce communication and data complexity in tree or mesh network topologies, a variety of *secure in-network aggregation* schemes have been proposed. In general, these schemes are able to compute arbitrary aggregation operations on data in a secure way. Unfortunately, in-network aggregation schemes have several drawbacks, which prevent their use in collective attestation protocols. For instance, they require to maintain a specific topology during aggregation [63, 117, 140, 149], are insecure upon corruption of multiple devices [63], can only estimate whether the aggregate has been manipulated [117, 149], or can only detect manipulations on the aggregate retrospectively after additional communication rounds [140, 149].

In contrast, *cryptographic aggregation* schemes provide rigorous security guarantees. They enable Message Authentication Codes (MACs) or digital signatures from multiple parties to be combined in a compact aggregate. Using the aggregate, a verifier can ensure that the data is authentic and indeed originates from the claimed sources. *MAC aggregation* schemes [77] are applicable to very resource constrained devices due to their high efficiency. However, they require each verifier to hold the (secret) MAC key of all parties, which makes them inapplicable in case verifiers are potentially untrustworthy. In comparison, *aggregate signature* schemes [22, 102] are less efficient, but enable verifiers to be untrustworthy, as they must only hold the public keys of all parties. *Multisignature* schemes [43, 70, 106] are a special case of aggregated signatures, where all parties are only allowed to sign the same data. Both aggregate and multisignature schemes are commonly based on RSA [70], discrete logarithms [16, 104, 106], pairings [22, 102], and lattices [43]. Schnorr multisignatures [16, 104, 138] are one of the simplest, most-well understood, and efficient multisignature schemes. On the downside, Schnorr multisignatures are generated in an interactive protocol, can only be aggregated at the time of signing, and are larger than some pairing-based signatures. Another drawback of both MAC and signature aggregation schemes is that they do not allow extracting (uncompress) individual MACs or signatures from an aggregate and also do not allow the aggregate to contain duplicates. Thus, two aggregates that both contain the same MAC or signature cannot be further compressed into a combined aggregate. Hence, existing schemes typically maintain a static tree-based overlay network for an efficient aggregation [138].

2.1.5 *Physical Attacks*

Classification. Physical attacks can be classified in three groups: *invasive*, *semi-invasive*, and *non-invasive* attacks [135]. Invasive attacks require an adversary to get access to the internal components of a targeted chip by decapsulating and deprocessing it. Next, the adversary can perform so-called microprobing attacks [136], in which needles are attached to the internal wiring of the chip. This enables the adversary to read out potential secrets or perform additional fault attacks. In fault attacks, the chip is manipulated to provoke errors and unintended states that can enable the adversary to access secrets or disable protection mechanisms. Invasive attacks are the most powerful attacks, but require expensive equipment, e.g., a microprobing station, highly skilled personnel, and much time. Similar to invasive attacks, semi-invasive attacks [133] require depackaging the chip in order to expose its surface. However, semi-invasive attacks rely on less expensive equipment and are easier to perform. This is because electrical contact to the internal lines of a chip is not required. Instead, laser scanning or thermal imaging is used to read out internal secrets, or fault attacks are performed based on ultraviolet radiation or optical fault injections. Nonetheless, both, invasive and semi-invasive, attacks require an adversary to capture the target device and analyze it for hours up to weeks in specialized laboratory environments [134].

Non-invasive attacks do not require physically tampering with the structure or packaging of a chip. Common non-invasive attacks are side-channel, brute force, fault injection, and data remanence attacks [135], which all use the chip as a black box. Non-invasive attacks are typically more dangerous than (semi-)invasive attacks. First, this is because non-invasive attacks leave no physical traces. Therefore, it is often unnoticed that a chip has been attacked and its assets have been compromised. Second, non-invasive attacks are easier to mount, as they require inexpensive equipment or even no equipment at all, as well as less knowledge and less time than (semi-)invasive attacks. On the other hand, non-invasive attacks can be prevented with less effort. In the past, many countermeasures against non-invasive attacks were proposed and applied in practice [121].

Countermeasures. The detection and mitigation of physical attacks on computer systems is an ever-present goal, which is commonly approached using tamper-evident and/or tamper-resistant hardware. While the former attempt to sense physical intrusions (e.g., by employing silicon-embedded sensors or sealing lacquer), the latter harden systems against tampering (e.g., by shielding hardware with solid materials). Standards like FIPS 140-2 [48] and PCI-HSM [35], as well as certain Common Criteria Protection Profiles [81] define different security levels with requirements on tamper-evident and tamper-resistant hardware. However, hardware that fulfills these security levels imposes significant costs in form of weight, space, power consumption, and money. Therefore, tamper-evident and tamper-resistant hardware is typically only applied to secure high-value assets, e.g., certificate authorities, but is not deployed in commodity embedded devices.

Absence detection constitutes a more general and lightweight technique to detect physical tampering. Instead of fully relying on hardware-based protection measures, absence detection builds on the assumption that an adversary must take a device offline for a noticeable amount of time to perform (semi-)invasive physical attacks [14]. Initially, absence detection was proposed to detect device failures [137]. Yet, later works applied absence detection to also defend against node capture attacks [32, 33]. In these attacks, an adversary physically captures a device to extract its cryptographic material or to reprogram and redeploy it. In order to detect node capture attacks, devices in the network collaboratively flag a device as captured if it fails to communicate with any other network device within a fixed time interval. This approach was extended in subsequent work [62] by statistical methods to detect absent neighbor devices in static network topologies. Although absence detection suffers from false positives in dynamic networks, it constitutes an effective solution to detect (semi-)invasive physical attacks on devices that lack tamper-evident and/or tamper-resistant hardware, e.g., commodity embedded devices.

2.2 SYSTEM MODEL

The main entity in our system model is the trustworthy network operator $\mathcal{O}$, who initializes and maintains a network with one or multiple embedded devices

$\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n$. The devices can be heterogeneous, which means that they may run different software and have distinct computational power, storage capacity, communication capabilities, and secure hardware. In particular, we consider low-end embedded devices that have very limited resources and capabilities. Based on the terminology for constrained devices from the Internet Engineering Task Force (IETF) [79], we define all embedded devices that have less than 100 MHz of computing power, 64 kB of RAM, and 256 kB of flash memory to be low-end devices. More powerful embedded devices are loosely referred to as mid-range and high-end devices. In addition, we assume that all devices whose integrity should be verifiable implement specific secure hardware features, which are defined in Section 2.3. During the operation of the network, $\mathcal{O}$ repeatedly wants to ensure the secure and safe operation of the maintained devices. To this end, $\mathcal{O}$ uses one of our proposed attestation protocols.

All our attestation protocols comprise a *deployment phase* that is executed by $\mathcal{O}$ prior to the actual attestation of devices. In the deployment phase, all devices in the network are initialized in a trustworthy environment. After deployment, $\mathcal{O}$ can repeatedly act as a *verifier* by executing the *attestation phase* of the respective protocol. This enables $\mathcal{O}$ to verify the integrity of one or multiple devices in the network, which act as *provers*. As a result of the attestation phase, $\mathcal{O}$ obtains an attestation report that indicates all prover devices whose software is trustworthy, i.e., unmanipulated and up-to-date. We refer to these devices as *healthy* or *uncompromised* devices, in contrast to *compromised* devices. In addition, our two protocols that defend against physical attacks (Chapters 6 and 7) further indicate the hardware integrity of prover devices, so that $\mathcal{O}$ can also ensure that prover devices are physically healthy. Moreover, our second protocol that defends against physical attacks (Chapter 7) enables not only the network operator $\mathcal{O}$, but also arbitrary other devices, to act as a verifier.

Throughout this thesis, we consider different communication technologies and network topologies, such as line, bus, star, tree, or mesh. In particular, we also regard ad hoc networks, in which devices cooperate to distribute data in the network, and thus form a decentralized and self-organized network. We consider networks that are static, where devices remain stationary, as well as dynamic, where devices can move freely. The latter are also referred to as MANETs or, in case devices only rarely have a connection to each other, as Delay-Tolerant Networks (DTNs). Nevertheless, we assume that the overall network topology remains connected and devices are only physically absent or offline for short times. The threshold how long devices are allowed to be unreachable depends on the particular attestation protocol. Note that such a threshold must exist because compromised devices can always refuse to respond to network messages, which makes it impossible to remotely distinguish between unreachable and compromised devices.

2.3 DEVICE REQUIREMENTS

Depending on the respective attestation protocol, prover devices must possess particular hardware security features to ensure a secure attestation. In general, all our protocols require prover devices to provide the minimal secure hardware features for remote attestation [51], which are common for all hardware-based attestation protocols. In Chapter 3, we present a solution to implement the required minimal secure hardware features on commodity low-end embedded devices, which is also employed for our use cases in Chapter 4. Nevertheless, in our subsequently proposed attestation protocols (Chapters 5, 6, and 7), we abstract from specific details on the implementation of the minimal secure hardware features on prover devices.

Furthermore, our attestation protocols that defend against physical attacks (Chapters 6 and 7) rely on additional assumptions. In specific, both protocols require prover devices to possess a write-protected Real-Time Clock (RTC) that is loosely synchronized between healthy devices. The RTC is needed to prevent malware from tampering with the device clock. Note that all existing protocols that defend against physical attacks [65, 68] rely on this assumption.

We henceforth refer to the execution space where all mentioned hardware properties for the respective attestation protocol are fulfilled as Trusted Execution Environment (TEE).

2.4 ADVERSARY MODEL

Software Adversary. We assume an adversary Adv_{sw}, who has full control over the communication medium (Dolev-Yao model [40]). Hence, Adv_{sw} is able to perform network attacks by eavesdropping, modifying, deleting, or synthesizing messages between devices. Furthermore, Adv_{sw} can perform software attacks, in which Adv_{sw} is able to compromise the software of all devices at will. On devices with a compromised software, Adv_{sw} has full control over the execution state, and can read from, or write to, any memory. However, Adv_{sw} is unable to bypass the hardware protection, and thus, cannot tamper with code, data, or the execution state in the TEE. DoS attacks are considered out of scope, since it is impossible to guarantee availability when Adv_{sw} is able to drop all messages.

Physical Adversary. We also consider a stronger adversary Adv_{hw}, who has the same capabilities as Adv_{sw} but can additionally physically attack devices in the network. After successfully tampering with a device physically, Adv_{hw} has full control over its clock as well as code and data in its TEE. However, common for absence detection protocols [32, 33, 68], we rely on the assumption that prover devices provide some form of physical tamper-resistance. The tamper-resistance requires Adv_{hw} to take a targeted device offline for a noticeable amount of time during the physical attack, e.g., to decapsulate the device [14, 135]. In general, physical attacks require not only much time, but also expensive equipment and

laborious handwork of highly skilled personnel. Depending on the hardware of provers and the capabilities of Adv_{hw}, we further assume that Adv_{hw} is only able to successfully tamper with a confined number of provers simultaneously and/or in total. In short, we assume that Adv_{hw} is limited in the following ways:

(1) **Mandatory:** Adv_{hw} must permanently take a prover device $\mathcal{P}_i$ offline for at least the *attack time* δ_a to physically compromise $\mathcal{P}_i$.

(2) **Optional:** Adv_{hw} is unable to physically compromise more than the *concurrency factor* β prover devices simultaneously.

(3) **Optional:** Adv_{hw} is unable to physically compromise more than the *attack limit* λ prover devices in total.

The parameters δ_a, β, and λ must be adjusted to the physical tamper-resistance of prover devices and the required security level. Note that the concurrency factor β and attack limit λ are optional limitations for Adv_{hw}, which only facilitate the detection of physical attacks. In practice, a low β increases the robustness of our protocol (Chapter 7), whereas a high λ enables a verifier to perform the attestation at arbitrary times, as opposed to periodically within short time intervals (Chapter 6 and 7). Both limitations can be disabled by setting β to ∞ and λ to 0. By contrast, the attack time δ_a is a mandatory limitation for Adv_{hw}. Thus, if δ_a is 0, our attestation protocols are unable to operate.

We acknowledge that non-invasive physical attacks, e.g., cache or power side-channels, may enable an adversary to bypass secure hardware during the operation of a device, i.e., without disabling the device. However, our protocols focus on the detection of invasive and semi-invasive attacks, which is why we expect that provers implement the various proposed mechanisms to prevent non-invasive attacks [121]. By contrast, invasive and semi-invasive attacks cannot be fully prevented by technical measures on the device itself. Performing invasive and semi-invasive attacks an adversary directly accesses the internal components of the target device, which requires at least the decapsulation of the device and potentially also the bypass of hardware tamper-resistance. Therefore, invasive and semi-invasive attacks require an adversary to take a device offline and analyze it with specialized laboratory equipment [14, 135].

Security Objectives. The security goals are dependent on the scope of the particular attestation protocol:

- Attestation protocols that focus on software attacks (Chapter 3, 4, and 5) are regarded as secure, if Adv_{sw} is unable to fake a healthy system state for a prover that is at the time of its attestation in a compromised software state.
- Attestation protocols that focus on software and physical attacks (Chapter 6 and 7) are regarded as secure, if Adv_{hw} is unable to fake a healthy system state for a prover that is at the time of its attestation in a compromised software and/or hardware state.

3 ATTESTATION OF COMMODITY LOW-END EMBEDDED SYSTEMS

In contrast to mid-range and high-end embedded systems, low-end embedded systems are particularly optimized towards low cost, small size, and low energy consumption. For this reason, low-end embedded devices offer only little computing power and few secure hardware features. These circumstances hamper the realization of secure attestation protocols for low-end embedded systems. In this chapter, we present ALE, an attestation protocol that is specifically designed for low-end embedded systems. ALE provides the same (strong) security guarantees as existing hardware-based attestation protocols while relying on less secure hardware features. In contrast to existing hardware-based protocols, this enables ALE to be applicable to a broad range of existing commodity low-end embedded devices. Furthermore, ALE is lightweight and efficient, which we demonstrate by evaluating our protocol on an exemplary commodity low-end embedded device.

Remarks. Parts of this chapter have been published in [85, 86, 127].

3.1 MOTIVATION AND CONTRIBUTION

Low-end embedded systems are characterized by their very constrained hardware, low energy consumption, small size, and low costs. Due to these characteristics, low-end embedded systems are widely deployed in many applications. For instance, low-end embedded devices are frequently used as basic sensors or actuators in safety-critical systems, e.g., road vehicles or industrial control systems. In these applications, malicious low-end embedded devices can cause significant physical damage, which is why it vital to ensure their security. However, the same characteristics that make low-end embedded devices so popular, prevent the implementation of effective security measures. For reasons of cost and space, low-end embedded devices lack cryptographic accelerators, common TEEs, e.g., ARM TrustZone or Intel SGX, or secure coprocessors, such as a TPM. This lack of secure hardware impedes the realization of secure attestation protocols.

Software-based attestation protocols [27, 98, 129] are independent of secure hardware and thus are applicable to low-end and legacy embedded systems. However, they provide questionable security guarantees, as they rely on assumptions that are hard to achieve in practice [10, 29]. In contrast, hardware-based attestation mechanisms provide much stronger security guarantees by relying on secure hardware that is built-in prover devices. As standardized and commercial secure hardware components like ARM TrustZone, TPM, Intel TXT, or Intel SGX are too complex and expensive to be used in low-end embedded systems, new security architectures, such as SMART [45], SANCUS [109], TrustLite [82], or

TyTan [23] have recently been proposed. Nevertheless, these architectures have only been implemented as prototypes and their future availability in commodity low-end embedded devices is uncertain.

Contribution. In this chapter, we present ALE, an attestation protocol for commodity low-end embedded systems. ALE combines the advantages of software-based and hardware-based attestation protocols, as it provides the same (strong) security guarantees as hardware-based protocols while being applicable to commodity low-end embedded devices, like software-based protocols. To detect compromised software, ALE measures the local software integrity during the boot process and immediately authenticates these measurements. Thus, a co-processor or TEE to securely store integrity measurements is no longer required. This enables ALE to rely on fewer secure hardware properties than previous protocols. In particular, our protocol only relies on write-protected boot code and a simple memory protection mechanism. We show that these minimal secure hardware properties are available on many existing low-end embedded devices. This makes our protocol applicable to a broad range of popular low-end embedded devices without requiring hardware modifications. Therefore, ALE can be retrofitted to many currently deployed systems. ALE's basic protocol only involves symmetric cryptography, which makes it lightweight and efficient. We implement our protocol on an exemplary low-end embedded device and demonstrate its practicality and efficiency.

Outline. In Section 3.2, we describe the secure hardware requirements of our attestation protocol and show that many existing low-end embedded devices provide these properties. Section 3.3 presents our attestation protocol. In Section 3.4, we describe two extensions that further increase the practicality and completeness of our protocol. Section 3.5 provides a proof of concept implementation and an evaluation of our protocol.

3.2 SECURE HARDWARE REQUIREMENTS

All remote attestation protocols build on two essential operations: During attestation, prover devices (i) measure their local software integrity, and (ii) authenticate their software integrity towards a verifier using a device-unique key. An adversary who can tamper with either of both operations is able to break the security of the respective attestation protocol. For this reason, all software and hardware that is involved in measuring and authenticating the software integrity constitutes the Trusted Computing Base (TCB) of attestation protocols.

As in any hardware-based attestation protocol, the TCB of our attestation protocol comprises secure hardware. In the following, we first specify our required hardware security properties. Afterwards, we explain their implementation on existing commodity low-end embedded devices.

3.2.1 *Hardware Security Properties*

Our attestation protocol demands the following hardware security properties from each prover device:

1. *Immutable Bootloader:* A write-protected memory region $\mathcal{B}$ that contains code and data and is immediately executed when the device starts;
2. *Secure Storage:* A device-dependent unique attestation key ak that is only accessible to the code in $\mathcal{B}$ and does not leak outside $\mathcal{B}$;
3. *Uninterruptible Execution:* Once code in $\mathcal{B}$ gets executed, execution cannot be interrupted until the control flow intentionally leaves $\mathcal{B}$.

In the next section, we discuss how these properties can be implemented on commodity low-end embedded devices. We will see that many existing devices provide hardware features that allow for the implementation of the properties.

3.2.2 *Implementation on Commodity Low-End Embedded Systems*

In the following, we demonstrate how each of the stated hardware security properties can be implemented on existing Commercial Off-The-Shelf (COTS) low-end embedded devices.

1st Property: Immutable Bootloader. Nowadays, it is common for commodity low-end embedded devices to provide a customizable bootloader that is stored protected in flash memory. On some devices, the flash memory can be separated into multiple sections that have dedicated lock bits for read protection, write protection, and also interrupt prevention [13]. Most commonly, the flash memory is divided into one bootloader section (BLS) and one application section. If the device at hand offers this feature, we propose that the BLS represents the memory region $\mathcal{B}$ and lock bits are set in a way that write access to the BLS is denied. This makes $\mathcal{B}$ immutable.

Other devices provide a more fine-grained flash protection, where memory regions of different sizes can be marked as read-only memory (ROM) or potentially also as execute-only memory (XOM) [141]. If the device at hand offers XOM or ROM, we propose to protect the boot code, which is immediately executed when the device starts, with the strongest supported memory protection available on the device, i.e., XOM if available and ROM otherwise. This memory region represents the bootloader $\mathcal{B}$. Note that once flash protection is set, it can only be unset by physically accessing the system. This process typically involves the erasure of the entire flash memory [13, 141].

2nd Property: Secure Storage. If the particular device offers a dedicated bootloader section (BLS) for $\mathcal{B}$, we suggest that the attestation key ak is stored in this memory section. Next, we propose to configure the lock bits in a way that read

access to $\mathcal{B}$ is denied if it is performed by code outside of the separated memory region [13]. Thus, ak can only be read during the execution of $\mathcal{B}$.

If the particular device does not support such lock bits, we suggest to store ak in a secure key storage whose access can intentionally be denied until the next device restart. In this way, code in $\mathcal{B}$ can read out ak once during device start and afterwards deny access to ak. As $\mathcal{B}$ is immutable and immediately gets executed when the device starts, an attacker is unable to access ak. A secure key storage which provides this functionality is, for instance, an SRAM PUF. Previous works have shown that the SRAM modules present in several low-end embedded devices can be used as PUF instances, so that cryptographic keys can be derived from the SRAM start-up values [78, 126]. Note that the start-up values can be deleted after they have been read out, so keys are only accessible at boot time. A further possible key storage is memory which provides the functionality to hide blocks, e.g., EEPROM block hide [141]. Once an EEPROM block is hidden, it is not accessible until the next restart of the device.

3rd Property: Uninterruptible Execution. If the device at hand provides a dedicated bootloader section (BLS) for $\mathcal{B}$, we suggest setting the lock bits of the bootloader section such that interrupts are denied during the execution of code in that section. On other devices, we propose to store both the interrupt vector table (IVT) and a default interrupt handler in write-protected memory (i.e., XOM or ROM). All interrupts in the IVT are configured to refer to the default interrupt handler. When an interrupt occurs and the default interrupt handler gets executed, it checks whether the interrupt was triggered during the execution of code in $\mathcal{B}$. If this is the case, the default interrupt handler denies interrupt processing. If this is not the case, the interrupt handler redirects execution to a user-defined interrupt handler which processes the particular interrupt [52]. A further approach is to let the default interrupt handler clean up sensitive data before control is handed over to the particular user-defined interrupt handler [142]. Both approaches impose no restrictions, since custom interrupts can still be deployed by modifying the user-defined interrupt handlers.

Summary. Table 3.1 provides an overview of popular low-end embedded development devices and shows, for each device, which hardware security properties can be achieved and how to achieve them. *BLB*, *BOOTRST*, *PALL*, etc. are the names of particular fuses and lock bits that have to be set to achieve the stated security. A checkmark or hyphen indicates the availability of a security feature. The overview illustrates that the required security features are available in many popular low-end embedded devices. This demonstrates the broad applicability of our attestation protocol.

3.3 ALE: ATTESTATION OF LOW-END EMBEDDED SYSTEMS

In this section, we present ALE, an attestation protocol for low-end embedded systems. The protocol consists of two different phases. In the *deployment phase*

Device	Immutable Code	Secure Storage	Uninterruptible Execution	Approach
panStamp AVR2	✓	✓	✓	$\mathcal{B}$ = BLS; BLB0=1; BLB1=4; BOOTRST=0
TinyDuino	✓	✓	✓	$\mathcal{B}$ = BLS; BLB0=1; BLB1=4; BOOTRST=0
Arduino UNO	✓	✓	✓	$\mathcal{B}$ = BLS; BLB0=1; BLB1=4; BOOTRST=0
RFduino	✓	✓	✓	Use MPU: Store $\mathcal{B}$ in R0; PALL=0; PR0=0
XinoRF	✓	✓	✓	$\mathcal{B}$ = BLS; BLB0=1; BLB1=4; BOOTRST=0
Ciseco	✓	✓	✓	$\mathcal{B}$ = BLS; BLB0=1; BLB1=4; BOOTRST=0
Pinoccio	✓	✓	✓	$\mathcal{B}$ = BLS; BLB0=1; BLB1=3; BOOTRST=0
Nanode	✓	✓	✓	$\mathcal{B}$ = BLS; BLB0=1; BLB1=4; BOOTRST=0
Arduino Yun	✓	✓	✓	$\mathcal{B}$ = BLS; BLB0=1; BLB1=4; BOOTRST=0
Libelium Wasmote	✓	✓	✓	$\mathcal{B}$ = BLS; BLB0=1; BLB1=4; BOOTRST=0
MICA2	✓	✓	✓	$\mathcal{B}$ = BLS; BLB0=1; BLB1=4; BOOTRST=0
Arduino M. 2560	✓	✓	✓	$\mathcal{B}$ = BLS; BLB0=1; BLB1=4; BOOTRST=0
Intel Quark D2000	✓	✓	✓	Explained in Section 3.5
TI Stellaris	✓	✓	✓	(1) $\mathcal{B}$ in ROM + EEPROMHIDE (2) $\mathcal{B}$ in ROM + SRAM PUF

Table 3.1: Overview of popular low-end embedded devices and their security features.

(Section 3.3.1) all devices are initialized by the network operator $\mathcal{O}$. Afterwards, $\mathcal{O}$ can repeatedly execute the *attestation phase* (Section 3.3.2) to verify the software integrity of a particular device in the network. Finally, we demonstrate the security of our protocol (Section 3.3.3).

3.3.1 *Deployment Phase*

In the deployment phase, the network operator $\mathcal{O}$ initializes all prover devices in a trustworthy environment. $\mathcal{O}$ deploys each device $\mathcal{D}_i$ with a unique attestation key ak_i that is stored in their secure storage, which is only accessible to the protected bootloader $\mathcal{B}$ (see Section 3.2). The bootloader $\mathcal{B}$ is deployed with the necessary code to measure the integrity of the installed software and authenticate the measurements with ak, as described in the next section. In addition, $\mathcal{O}$ equips

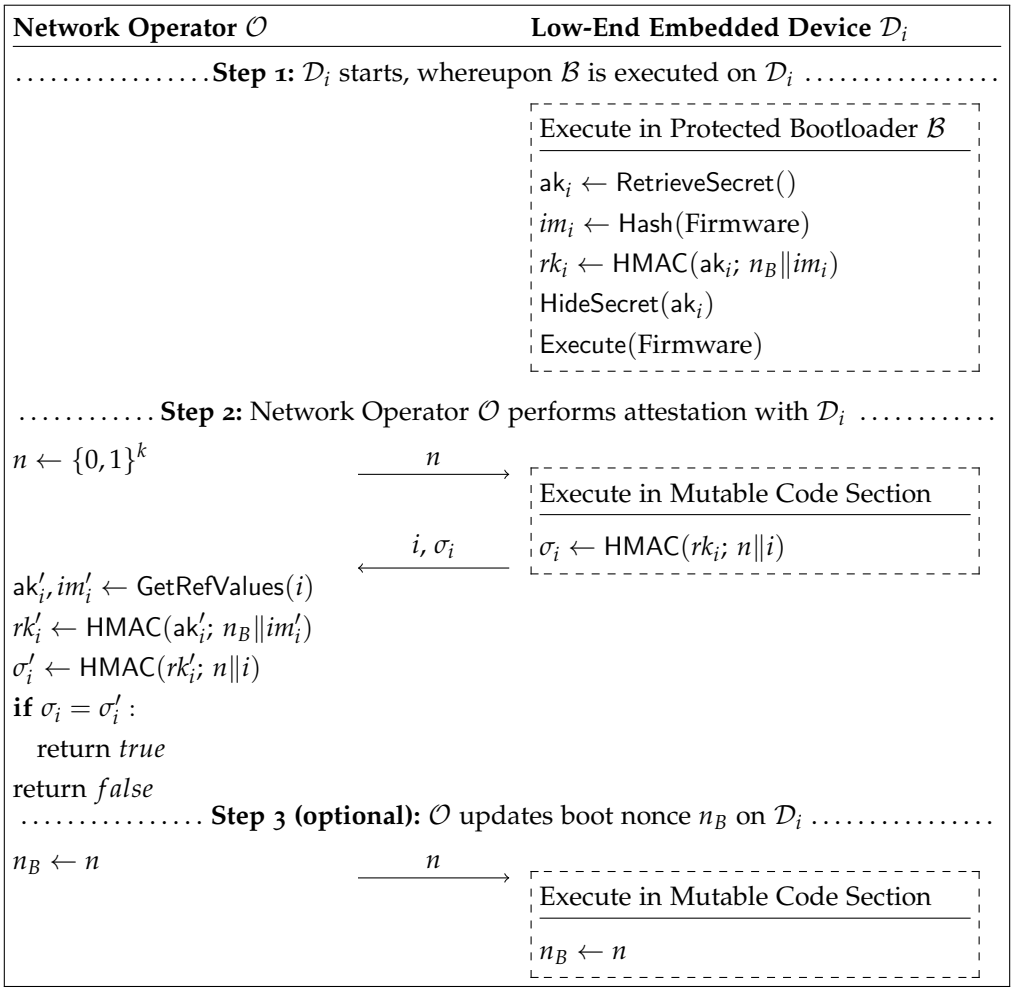


Figure 3.1: Attestation of a low-end embedded device $\mathcal{D}_i$ by the network operator $\mathcal{O}$.

each device $\mathcal{D}_i$ with a unique identifier i and a random boot nonce n_B, which are stored in mutable (unprotected) memory, outside of $\mathcal{B}$. The code to interact with $\mathcal{O}$ is also stored outside of $\mathcal{B}$ in mutable memory, which significantly reduces the size of $\mathcal{B}$. This minimizes our TCB, as shown in Section 3.5. For each initialized prover device, the operator $\mathcal{O}$ stores its identity, attestation key, and known-good integrity measurement of the respective software.

3.3.2 *Attestation Phase*

The attestation phase consists of three steps, which are illustrated in Figure 3.1. In the first step, devices boot and measure their software integrity. In the second step, the network operator verifies the integrity of devices. The third step can be executed optionally and allows the operator to update the boot nonce of devices. Updating the boot nonce enables a recovery from runtime attacks, as we

explain in Section 3.3.3. All three steps are described in detail in the following paragraphs.

Step 1. When a device $\mathcal{D}_i$ starts, it immediately executes its protected bootloader $\mathcal{B}$. The first operation performed by $\mathcal{B}$ is retrieving the secret attestation key ak_i from the respective secure storage. Next, the installed firmware is measured, which involves computing a hash value over the memory region where the firmware is stored. The memory regions to be measured are either predefined in $\mathcal{B}$ or are read out from a software certificate (see Section 4.2). The resulting integrity measurement im_i is stored outside of $\mathcal{B}$. Next, $\mathcal{D}_i$ generates its so-called *response key* rk_i. To this end, $\mathcal{D}_i$ computes an Hash-based Message Authentication Code (HMAC) with its attestation key ak_i over the boot nonce n_B and the integrity measurement im_i. Afterwards, $\mathcal{D}_i$ ensures that no information about ak_i are leaked. As explained in Section 3.2, this may involve erasing certain values in memory or executing specific instructions to lock ak_i against further access. Finally, the control-flow leaves $\mathcal{B}$, as the measured firmware is executed.

Step 2. In the second step, the network operator $\mathcal{O}$ generates an attestation challenge, which consists of a fresh nonce n. The attestation challenge is sent to the device $\mathcal{D}_i$ that $\mathcal{O}$ wants to verify. Receiving the challenge, $\mathcal{D}_i$ computes its attestation response, which contains the identifier i of $\mathcal{D}_i$ as well as an HMAC σ_i that is computed with rk_i over n and i. Afterwards, $\mathcal{D}_i$ sends $\mathcal{O}$ its attestation response, i.e., its identity i and HMAC σ_i.

In order to verify the attestation response, $\mathcal{O}$ computes the expected (known-good) HMAC and compares it with the received HMAC σ_i. To this end, $\mathcal{O}$ first uses the received identifier i to retrieve the known-good attestation key ak'_i and integrity measurement im'_i of $\mathcal{D}_i$. Next, $\mathcal{O}$ computes the expected response key rk'_i and the expected HMAC σ'_i, as detailed in Figure 3.1. Finally, $\mathcal{O}$ compares the computed HMAC σ'_i with the received HMAC σ_i. If both values match, $\mathcal{O}$ considers $\mathcal{D}_i$ to be in a trustworthy software state. If not, $\mathcal{D}_i$ is regarded as compromised. We would like to point out that the resulting HMAC σ_i of $\mathcal{D}_i$ can also be useful for purposes other than attestation. For instance, $\mathcal{D}_i$ can additionally compute the HMAC σ_i over particular data to be authenticated, which is then send to $\mathcal{O}$. This enables $\mathcal{O}$ to verify whether received data indeed originates from $\mathcal{D}_i$.

Depending on the use case, $\mathcal{O}$ may repeat step 2 at later stages, continue with step 3, or instruct $\mathcal{D}_i$ to restart and execute step 1 again.

Step 3. In certain use cases, the network operator $\mathcal{O}$ may additionally execute step 3 to update the boot nonce of a particular prover device $\mathcal{D}_i$. Performing step 3, $\mathcal{O}$ sends $\mathcal{D}_i$ an update message, which contains the nonce n generated in step 2. Afterwards, both $\mathcal{O}$ and $\mathcal{D}_i$ overwrite the boot nonce n_B with n. Updating n_B allows recovering from attacks in which Adv_{sw} compromised a device $\mathcal{D}_i$ at runtime and obtained its response key rk_i. This is because a new n_B causes rk_i to change in the next attestation phase. Thus, Adv_{sw} is unable to use the obtained rk_i to forge a valid system state for $\mathcal{D}_i$ in subsequent runs of the attestation (see Section 3.3.3). Note that n_B may also be updated as part of the attestation request

in step 2, so that $\mathcal{O}$ need not send a dedicated update message. We refer to our automotive attestation use case in Section 4.1 for details on updating n_B in step 2.

3.3.3 *Security Analysis*

In the following, we analyze the security of ALE against the software adversary Adv_{sw} (Section 2.4). We show that our protocol is able to achieve the same security guarantees as all hardware-based load-time attestation protocols, which includes protocols based on the widely-deployed TPM. To this end, we analyze the capabilities of Adv_{sw} to perform network attacks, load-time software attacks, and runtime software attacks. Recall that we consider our attestation protocol as secure, if Adv_{sw} is unable to pass the attestation with a compromised device.

Network Attacks. By performing network attacks, Adv_{sw} is unable to obtain the response key rk of prover devices, as rk never leaves devices. However, without knowledge of rk, Adv_{sw} is unable to bypass the challenge-response protocol that is executed in step 2. To break the challenge-response protocol, Adv_{sw} would need to generate an HMAC σ_A that equals $\sigma = \mathsf{HMAC}(rk_i; n \| i)$ with a random nonce n and an arbitrary identifier i. Under the assumption that the HMAC is cryptographically secure, Adv_{sw} is unable to forge such a σ_A without knowing rk. Replaying recorded HMACs from previous attestation phases or other prover devices is also unpromising for Adv_{sw}. This is because the network operator $\mathcal{O}$ always chooses a fresh nonce n as a challenge, which results in a unique HMAC σ that is valid in each run of the challenge-response protocol. In addition, each prover device $\mathcal{D}_i$ holds a device-dependent attestation key ak_i, such that given a challenge n, the response HMAC of different devices is never identical. Finally, by dropping or manipulating challenge messages and response messages, Adv_{sw} can only perform DoS attacks, but is unable to subvert the security of our attestation protocol. Nevertheless, in practice, it may be useful for the network operator $\mathcal{O}$ to authenticate messages, in order to prevent Adv_{sw} from impersonating $\mathcal{O}$. For instance, this would prevent Adv_{sw} from performing simple DoS attacks, in which Adv_{sw} distributes bogus boot nonces to devices in step 3.

Load-Time Software Attacks. Adv_{sw} may compromise the software of a prover device before it is loaded and thus measured. To this end, Adv_{sw} may have physical access to the prover device, which allows Adv_{sw} to reprogram the software. Executing a compromised software gives Adv_{sw} access to all readable and writable memory as well as complete control over the execution environment. However, Adv_{sw} cannot bypass the secure hardware, meaning that Adv_{sw} is unable to violate any of the three hardware security properties (Section 3.2.1). In particular, Adv_{sw} is unable to prevent the execution of the protected bootloader $\mathcal{B}$ at the start of the prover device (1st property). Thus, Adv_{sw} cannot interfere with the execution of step 1 of our attestation protocol.

In step 1, the installed software is measured. In addition, the response key rk is derived from the measurements and the secret attestation key ak. On devices

with a compromised software, the integrity measurements *im* deviate from its known-good integrity measurements *im'*. Consequently, the derived *rk* does not match the proper *rk'* that $\mathcal{O}$ uses in step 2 to verify the attestation response. Adv_{sw} is unable to compute the proper *rk*, since Adv_{sw} lacks the attestation key ak. This is because ak is only accessible to the bootloader code (2nd property) whose execution is uninterruptible (3rd property). Note that ak is retrieved at the start of $\mathcal{B}$ and hidden before the execution leaves $\mathcal{B}$. Hence, the proper *rk* is only generated if $\mathcal{B}$ is executed from the beginning, which prevents Adv_{sw} from obtaining *rk* by generating a valid *im* in memory and then jumping to the computation of *rk* in the middle of $\mathcal{B}$. As a result, Adv_{sw} is unable to obtain ak and compute *rk*, or make $\mathcal{B}$ leak *rk*. Yet, without knowledge of *rk*, Adv_{sw} is unable to bypass our attestation protocol, as explained in the previous paragraph.

Runtime Software Attacks. Adv_{sw} may also compromise the software of a prover device at runtime, e.g., by exploiting a vulnerability that allows arbitrary code execution. Since runtime attacks are performed after step 1, where the software is measured, they enable Adv_{sw} to access the proper response key *rk* in memory. With the proper *rk*, Adv_{sw} is able to compute valid attestation responses when the network operator $\mathcal{O}$ performs step 2 with the compromised device. In this way, Adv_{sw} can pass the attestation with a compromised device, i.e., bypass the security of our attestation protocol. Nevertheless, we would like to point out that all load-time attestation protocols, including the TPM, are vulnerable to runtime attacks. Furthermore, even dynamic attestation protocols are unable to detect runtime attacks in case they are performed after the software is measured.

To recover from runtime attacks, $\mathcal{O}$ either needs to install a new software on the prover device or execute step 3 of our attestation protocol. Afterwards, the prover device needs to reboot. Installing a new software leads to a new integrity measurement *im* and executing step 3 updates the boot nonce n_B. Since the response key *rk* is dependent on both *im* and n_B, both measures cause *rk* to change in the subsequent execution of step 1. Thus, a new *rk* is valid in step 2, so that the response key which Adv_{sw} obtained in the runtime attack is of no value. Hence, Adv_{sw} would need to perform a runtime attack again to bypass the security of our protocol. Yet, in the meantime, a software update could have fixed the original vulnerability that allowed Adv_{sw} to perform runtime attacks.

3.4 EXTENSIONS FOR REAL-WORLD USE

In the following, we discuss extensions to ALE that are valuable for real-world applications. The first extension provides support for provisioning an attestation key and requires a slight extension of our hardware requirements. The second extension enables untrusted third parties to act as a verifier and perform the attestation of prover devices.

Attestation Key Provisioning. In cases where the attestation key ak of a prover device may have been potentially compromised, it is desirable to provision a

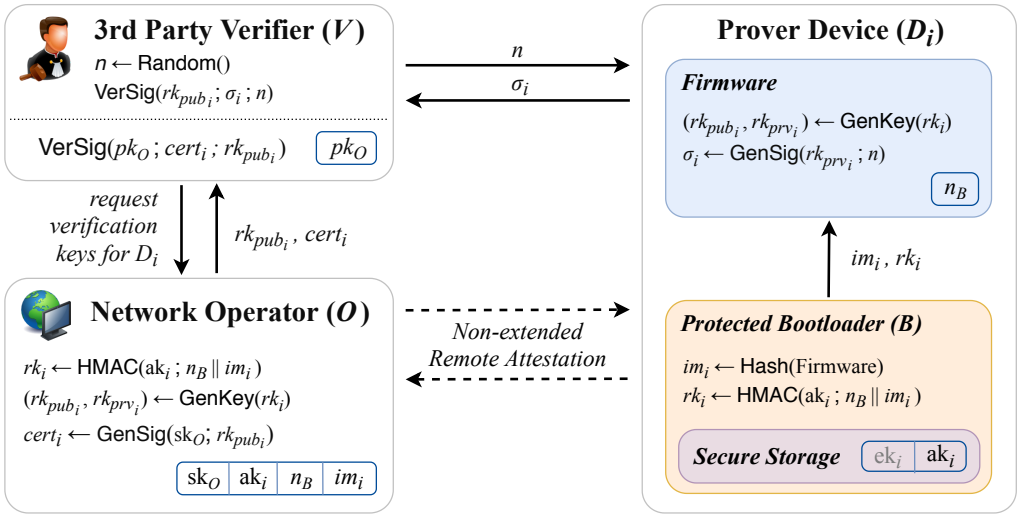


Figure 3.2: Illustration of third-party verification. The third-party $\mathcal{V}$ first obtains the public verification key rk_{pub_i} from $\mathcal{O}$ and then verifies the prover device $\mathcal{D}_i$.

new ak to the prover device. Examples include a user who takes ownership of a new or second-hand device, or a corrupted verifier who potentially leaked ak. To provision a new ak, we follow the TCG concept and propose that each prover device $\mathcal{D}_i$ is equipped with a device-unique *endorsement key* ek_i during its production. The endorsement key ek_i is a symmetric key that is only shared between the device $\mathcal{D}_i$ and the manufacturer. This allows the manufacturer to authorize key provisioning requests for ak from the network operator $\mathcal{O}$ to $\mathcal{D}_i$. At the same time, $\mathcal{O}$ can ensure the authenticity of the provisioning target $\mathcal{D}_i$.

Provisioning ak_i based on ek_i can be realized by implementing a standard key exchange or key transport protocol in the protected bootloader $\mathcal{B}$. Since a key exchange based on a shared secret is a well-known technique in cryptography, we omit the description of a detailed instantiation. Nevertheless, we like to explain the slightly extended hardware requirement for supporting the key provisioning of ak_i. In particular, the provisioning requires the secure storage for ak_i to be writable by the protected bootloader $\mathcal{B}$, but protected against read and write access by any other code outside $\mathcal{B}$. In addition, ek_i must be protected against read access by code outside $\mathcal{B}$, which resembles the previously defined exclusive access for ak_i (see Section 3.2.1).

Third-Party Verification. In many use cases, low-end embedded devices operate not only with a single trusted entity, but establish a variety of interactions with user platforms, infrastructure components, and cloud backends. However, as the attestation key ak is a critical asset during attestation, sharing ak with all possible verifiers would impose a security risk. To address this issue, we extend our protocol to enable potentially untrusted entities to act as verifiers.

For this purpose, we turn the original network operator $\mathcal{O}$ into a Certification Authority ($\mathcal{CA}$). The idea is that a prover device $\mathcal{D}_i$ no longer uses its response

key rk_i to compute attestation responses. Instead, $\mathcal{D}_i$ derives a signature key pair based on the pseudo-random input of rk_i, whose private part is then used to compute the attestation response. In order that a third-party verifier $\mathcal{V}$ can perform the attestation with $\mathcal{D}_i$, only the public signature key computed from rk_i must be shared with $\mathcal{V}$. This public signature key is also certified by $\mathcal{O}$, which facilitates its secure distribution, e.g., in case of key updates. Thus, only ak_i needs to be stored in a secure environment at $\mathcal{O}$, whereas the key to verify attestation response is public. The detailed protocol is illustrated in Figure 3.2 and described in the following.

To enable a third-party verification, we propose to modify step 2 of our attestation protocol. After a prover device $\mathcal{D}_i$ executed step 1, as described in Section 3.3.2, it uses its response key rk_i to generate a signature key pair $(rk_{pub_i}, rk_{prv_i}) \leftarrow \mathsf{GenKey}(rk_i)$. The signature key pair is generated deterministically, for example, using ECC key generation with rk_i as a seed value. As the network operator $\mathcal{O}$ knows $\mathcal{D}_i$'s attestation key ak_i, integrity measurement im_i, and boot nonce n_B, $\mathcal{O}$ is able to reproduce rk_i and thus also rk_{pub_i} and rk_{prv_i}. This enables $\mathcal{O}$ to issue a certificate $cert_i$ that contains a signature from $\mathcal{O}$ over rk_{pub_i} and thus certifies the authenticity of rk_{pub_i}.

A third party verifier $\mathcal{V}$ initially queries the network operator for $\mathcal{D}_i$'s public response key rk_{pub_i} and the corresponding certificate $cert_i$. Using the certificate, $\mathcal{V}$ then verifies the authenticity of rk_{pub_i}. Upon a successful verification, $\mathcal{V}$ is ready to verify the software integrity of $\mathcal{D}_i$. To this end, $\mathcal{V}$ challenges $\mathcal{D}_i$ with a fresh nonce n. Subsequently, $\mathcal{D}_i$ generates a signature σ_i of n with rk_{prv_i} by computing $\sigma_i \leftarrow \mathsf{GenSig}(rk_{prv_i}; n)$. After $\mathcal{D}_i$ sends σ_i to $\mathcal{V}$, $\mathcal{V}$ uses rk_{pub_i} to verify σ_i. If the verification succeeds, i.e., $\mathsf{VerSig}(rk_{pub_i}; \sigma_i; n) = true$, $\mathcal{V}$ assumes that $\mathcal{D}_i$ is in a trustworthy software state, otherwise not. Since only public information are required to verify the software integrity of $\mathcal{D}_i$, $\mathcal{V}$ can be potentially untrusted. Note that each time the response key rk_i of $\mathcal{D}_i$ changes, e.g., as $\mathcal{D}_i$ is supplied with a new software or a new boot nonce, the public response key rk_{pub_i} also changes, so that $\mathcal{V}$ needs to obtain and verify the new rk_{pub_i} from $\mathcal{O}$.

3.5 PROOF OF CONCEPT

As a target platform for the implementation of ALE, we selected the Intel Quark D2000. Our implementation comprises measuring the installed firmware, deriving the response key *rk*, and managing the hardware protection for ak and ek. We show that the implementation of our protocol is very efficient, with a code and data footprint in the range of 4 kB, and a boot delay of less than 200 ms. We refer to our attestation applications (Chapter 4) for an evaluation of ALE on ARM-based platforms as well as an evaluation that also involves network communication and regards the verifier's side.

Component	Size (Bytes)		Runtime (ms)	
	`-Os`	`-O1`	`-Os`	`-O1`
Base ROM	1955	2115	6.11	5.93
Protected Bootloader $\mathcal{B}$				
Base Logic	168	193	< 0.01	< 0.01
HMAC-SHA-256 (32 B)	1819	2061	1.54	1.44
HMAC-SHA-256 (32 kB)	1819	2061	148.23	145.37
ak Protection	295	337	0.02	0.02
ek Protection	378	448	< 0.01	< 0.01

Table 3.2: Implementation overhead for the Intel D2000 (x86). The memory footprint in bytes is shown left and the runtime overhead in milliseconds is shown right, both with compiler optimizations for size (`-Os`) and runtime (`-O1`).

3.5.1 *Implementation*

The Intel Quark microcontroller D2000 employs an x86 Quark CPU that operates at 32 MHz, 8 kB SRAM, 32 kB main flash memory, as well as two regions of One-Time-Programmable (OTP) flash memory (4 kB and 4 kB). The Intel D2000 is tailored towards scenarios where low energy consumption is required. We use the Intel Quark Microcontroller Software Interface (QMSI) [100] and Zephyr RTOS [101] as a firmware stack. In the following, we describe the implementation of our required hardware security properties (Section 3.2.1) on the D2000.

Immutable and Uninterruptible Bootloader (1st and 3rd property). The D2000 boots directly from an 8 kB OTP flash partition. A hardware-enforced OTP lock permanently disables write accesses to the OTP partition of the flash memory. It further deactivates the mass erase capability of the OPT partition and at the same time disables JTAG debug access. Locking the OTP partition is done by setting bit '0' at offset `0x0` of the flash memory region to '0'.

Secure Storage (2nd property). We store ak in main flash to support updates via key provisioning. One of the D2000 Flash Protection Regions (FPR) is set up and locked by the bootloader $\mathcal{B}$ to prevent read access by later firmware stages. In order to store the long-term key ek, we use the OTP flash region of the D2000. The 8 kB OTP supports read-locking of the upper and lower 4 kB regions of OTP flash. As this read protection also inhibits execute access, we store ek at the upper end of the OTP memory and set the read-lock just at the very end of bootloader execution. The read-lock for the lower and upper OTP region is activated by programming the bits `ROM_RD_DIS_L` and `ROM_RD_DIS_U` of the `CTRL` register.

3.5.2 *Performance Evaluation*

In the following, we evaluate the memory and runtime overhead of our attestation protocol. We present measurements for the standard software stack and the protected bootloader $\mathcal{B}$. Numbers for $\mathcal{B}$ are further separated with respect to the base logic (data structures, memory management), cryptographic operations, and key protection logic. For memory footprint and runtime, we provide measurements for code that was optimized for size (`-Os`) and code that was optimized for runtime (`-O1`) by the compiler.

Memory. To measure the memory footprint, we counted all static code segments (`.text`) and read-only data segments (`.rodata`) of the firmware. Table 3.2 lists results for code that is optimized towards size (`-Os`) or runtime (`-O1`). On the Intel D2000, the protected bootloader $\mathcal{B}$ consumes 2.6 kB on top of the QMSI stock ROM of 2 kB. This fits well within the total 8 kB available for bootloader code. The application flash is left for potential application code, except for a small part that is reserved for ak storage.

Runtime. The main runtime overhead of our attestation protocol is incurred by the cryptographic operations, as shown on the right side of Table 3.2. More specifically, the main overhead is caused by the computation of the HMAC, whose precise runtime is dependent on its input size. Runtime measurements of the HMAC are given for a small memory block of 32 B to compute the response key, and a large memory region of 32 kB to reflect the software integrity measurement. The D2000 is fast in computing authenticated measurements over various memory regions due to its quick flash access. In particular, the D2000 requires only 145 ms for hashing 32 kB. By contrast, the key protection entails a negligible runtime, as it takes only 0.03 ms in the worst-case. As a reference, booting the unmodified base ROM, without our attestation protocol, takes on average 6 ms.

3.6 SUMMARY

In this chapter, we presented ALE, an attestation protocol that is particularly suited for low-end embedded systems. Existing hardware-based attestation protocols typically rely on a full-featured TEE to measure the software integrity and securely store the measurements. ALE relaxes these requirements by (i) enforcing that integrity measurements are performed at device start before malicious code can be executed, and (ii) immediately authenticating all measurements. This way, our attestation protocol is able to provide the same (high) security guarantees as other hardware-based protocols while relying on less secure hardware. In particular, this makes ALE applicable to a broad range of existing low-end embedded devices. Additionally, our basic protocol only relies on symmetric cryptography, which enables a low memory footprint and a low runtime overhead. In detail, measurements on an Intel Quark D2000 with 32 MHz showed that our protocol entails a runtime overhead of 200 ms and a memory footprint of 4 kB.

ALE's underlying mechanism for a secure attestation of low-end embedded devices can be used in other attestation protocols proposed in this thesis. In the next chapter, ALE is applied in two particular use cases (Chapter 4).

4 ATTESTATION APPLICATIONS WITH EMBEDDED SYSTEMS

Remote attestation is a versatile security service, which can be useful in many applications. This chapter presents two specific applications for the attestation of embedded systems. First, we apply attestation in the automotive context (Section 4.1). More specifically, we present a solution that uses attestation to ensure the secure operation of embedded systems in road vehicles. Thus, our solution prevents compromised embedded systems from jeopardizing the safety of vehicles. In the second application (Section 4.2), we combine remote attestation with existing code update techniques to achieve verifiable code updates, which enforce the proper installation of software as well as the removal of malware. This way, trust can be reestablished in potentially compromised devices, e.g., devices with a (previously) vulnerable software that may have been compromised by an adversary. Our solutions for both applications build on our attestation protocol for low-end embedded systems (Chapter 3), which demonstrates the protocol's applicability in practice.

Remarks. Parts of this chapter have been published in [85, 86].

4.1 USE CASE 1: ELECTRONIC CONTROL UNITS IN ROAD VEHICLES

4.1.1 *Motivation and Contribution*

In the past decades, vehicles converted from mostly mechanical systems to complex electronic systems with dozens of interconnected embedded computers, called Electronic Control Units (ECUs). Nowadays, ECUs are accessible through various communication protocols and interfaces, of which some are even wireless, e.g., Bluetooth, WiFi, and LTE. This evolution drastically increased the attack surface of vehicles and rendered established techniques to ensure the safety of cars, like redundancy, reliability, and determinism, insufficient. In fact, recent attacks [107] and security evaluations [30, 50] have demonstrated that ECUs can be manipulated to perform malicious actions, which can lead to life-threatening accidents. Consequently, the safety of modern vehicles can only be guaranteed by establishing an appropriate level of security.

To secure vehicles, much effort has gone into the development [112, 119] and standardization [2] of authentication protocols. These protocols protect against network attacks, in which the adversary attempts to forge or manipulate messages in the automotive network. Although authentication protocols are essential for security, they cannot protect against all attacks. In particular, they are unable to defend against software attacks, in which the adversary compromises the software of ECUs to perform malicious actions. For instance, the firmware

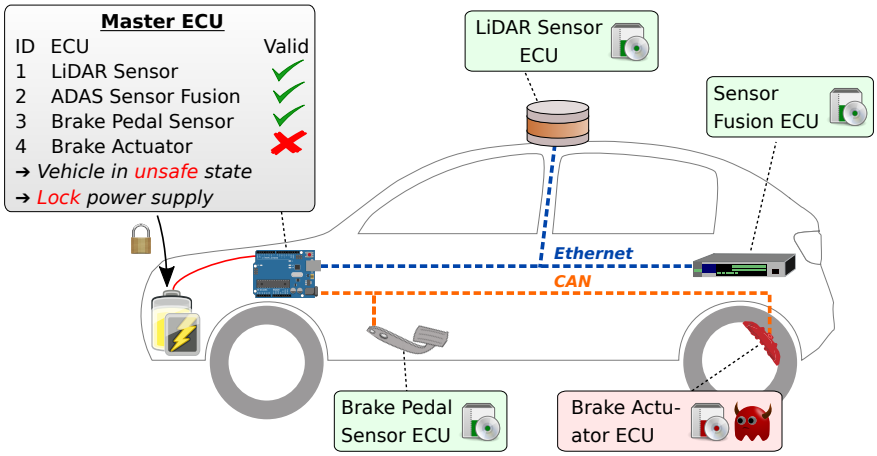


Figure 4.1: Illustration of our system architecture in an electric car.

of brake actuators may be manipulated to refuse applying the actual brakes (cf. Figure 4.1). As already explained in previous chapters, remote attestation is a key technology to defend against such software attacks.

The potential of remote attestation to protect vehicles against software attacks has already been recognized in existing works [97, 113, 145]. However, existing solutions suffer from limited applicability, as they either require ECUs to implement a Trusted Platform Module (TPM) [97, 113] or the SANCUS research architecture [145]. Yet, ECUs are not equipped with TPMs and SANCUS is commercially unavailable. Furthermore, existing works either lack an implementation and evaluation of the attestation [97, 145], or only provide an evaluation on personal computers, but not on embedded systems (i.e., actual ECUs) [113].

Contribution. In this section, we present a scheme that recurrently ensures the safe and secure operation of ECUs in road vehicles. In our scheme, a trusted master ECU verifies the software integrity of all safety-critical ECUs in the vehicle, each time drivers or passengers unlock the vehicle and open the door(s). Only after all safety-critical ECUs are successfully verified, the master ECU mechanically allows the vehicle to move, as illustrated in Figure 4.1. This way, compromised ECUs are prevented from jeopardizing the safety of the vehicle.

To address the heterogeneous character of ECUs in road vehicles, our scheme comprises two attestation techniques. The first technique is suited for simple ECUs that possess only a small and modest system architecture, such as basic sensors and actuators. It is based on our attestation protocol for low-end embedded devices (Chapter 3). The second technique targets advanced ECUs that feature ARM application processors, and hence are suited for more complex tasks, e.g., sensor fusion. It makes use of the ARM TrustZone technology to securely measure installed software and report the measurements. Both techniques are applicable to a broad range of existing commodity ECUs that are deployed in road vehicles.

We implement our scheme on commodity devices, which we connect in an automotive CAN and Ethernet network. Measurements conducted in our setup demonstrate that 100 ECUs can be verified in less than 1.4 seconds. Note that typical vehicles contain 70-100 ECUs in total, so that the amount of safety-critical ECUs can be expected to be less than 100. Based on our measurement results, we argue that drivers and passengers are unable to notice any overhead to the startup time of vehicles.

Outline. This section is organized as follows. First, we present our attestation scheme (Section 4.1.2). Next, we evaluate our scheme (Section 4.1.3). Finally, we conclude (Section 4.1.4).

4.1.2 *Attestation Scheme for Road Vehicles*

We first describe the basic functioning of our attestation scheme and then explain its technical implementation details on simple and advanced ECUs.

Overview

The architecture of our scheme is analogous to ALE, our protocol for low-end embedded devices (Chapter 3), except for two minor changes: The vehicle manufacturer takes over the role of the network operator $\mathcal{O}$ in the deployment phase, and the master ECU takes over the role of $\mathcal{O}$ in the attestation phase.

Accordingly, the center of our attestation scheme constitutes the master ECU. At each time the vehicle is unlocked and then opened, the master verifies the software integrity of a predefined set of safety-critical ECUs by executing our proposed attestation protocol. The set of safety-critical ECUs is fixed at the roll-out of the vehicle by the manufacturer, but can be changed later on, e.g., in case ECUs need to be replaced. The manufacturer deploys each safety-critical ECU with two different nonces, n_B and n, a unique identifier id, and a unique attestation key ak that is only known to the ECU and the master. Furthermore, the master and all safety-critical ECUs are supplied with the protocol code and data, as detailed in Section 4.1.2. The protocol is illustrated in Figure 4.2 and described in the following. The attestation takes place in two steps that are sequentially executed when the vehicle is unlocked and opened. Note that compared with ALE, we merged step 2 and 3, which decreased the number of steps by one.

Step (1). All safety-critical ECUs boot and measure their local software integrity. To this end, ECUs compute a hash value over their installed software and store it in im, the integrity measurement variable. Afterwards, each ECU derives a so-called response key rk. ECUs generate rk by computing an Hash-based Message Authentication Code (HMAC) over the boot nonce n_B and the measurements im with the attestation key ak. In step (2), rk is used to answer the attestation challenge from the master. If an ECU is in a compromised software state, im differs from the ECU's known good integrity measurement, so that rk does not

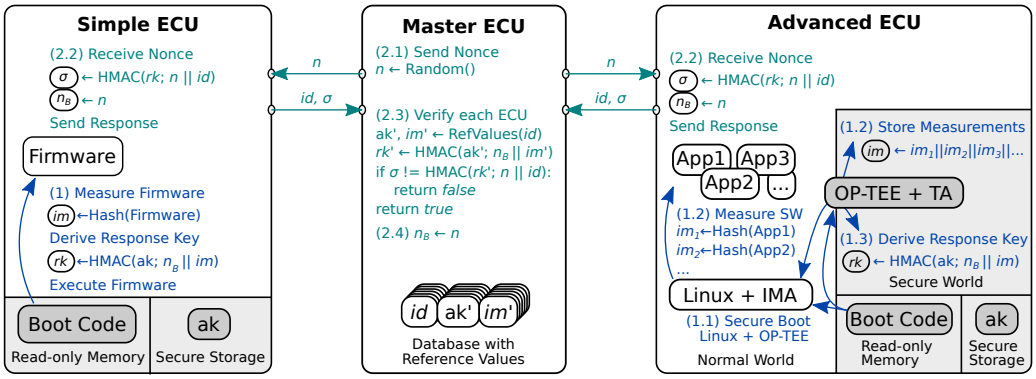


Figure 4.2: Attestation of simple and advanced ECUs by the master ECU. Code and data that is protected by secure hardware is shown in gray.

match the ECU's expected response key. In step (2), this is eventually detected by the master ECU when verifying the attestation response.

For a secure attestation, it is crucial that an adversary Adv_{sw} is unable to obtain access to the attestation key ak of ECUs, and cannot tamper with the computation of im and rk. The details to ensure this depend on the ECU type and are subject to the following subsection (Section 4.1.2).

Step (2). The master ECU generates a random nonce n and broadcasts it to all safety-critical ECUs. The nonce n functions as a challenge and prevents Adv_{sw} from performing replay attacks with recorded attestation responses.

Upon the reception of n, ECUs overwrite n_B with n. Since rk is dependent on n_B, this causes rk to change in each attestation run. As discussed in Chapter 3, this enables our scheme to recover from runtime attacks, in which Adv_{sw} compromises the software of ECUs at runtime, i.e., after step (1). Next, ECUs generate their attestation response, which consists of two parts: (i) the identity id of the ECU, and (ii) an HMAC σ that is computed over n and id with the response key rk from step (1). Finally, ECUs send their attestation response, i.e., id and σ, to the master ECU.

The master ECU receives all responses and verifies them as follows. For each safety-critical ECU, the master stores the known good integrity measurement im' as well as the secret attestation key ak'. Based on the id contained in a received response, the master retrieves the known good integrity measurement im' and the attestation key ak' of an ECU. Using ak' and im' as well as the previous nonce n_B, the master then computes the expected response key rk'. Afterwards, the master generates an HMAC over n and id with the computed rk', and compares it with σ from the attestation response. If both values match, the attestation response of the particular ECU is considered valid.

Only if all safety-critical ECUs replied with a valid response, the master ECU regards the vehicle to be in a safe state. Only then, the master enables the vehicle to move by releasing the lock for the ignition switch and/or power supply. Hence,

compromised ECUs can only prevent the vehicle from moving, but are unable to jeopardize its safety.

Technical Details

Security Requirements. For a secure attestation, it is crucial that an adversary Adv_{sw} (i) is unable to access ak, and (ii) cannot manipulate the computation of the software integrity measurements *im* and the response key *rk*. If (i) and (ii) are ensured, Adv_{sw} cannot compute *rk*. In addition, *rk* will reflect the software integrity of the ECU. This means that *rk* only matches the expected response key *rk'*, which is used by the master to verify the attestation response, in case the respective ECU is in a trustworthy software state. Since Adv_{sw} can only bypass our attestation scheme by violating (i) or (ii), all hardware and software that ensures (i) and (ii) is our Trusted Computing Base (TCB).

More specifically, our TCB software consists of all code that is executed in step (1), in which ak is accessed and *im* and *rk* are computed. Common for hardware-based attestation schemes, we require the TCB software to be supported by secure hardware that must enable the implementation of the following well-known properties [51] (cf. Section 3.2.1): (i) TCB code has exclusive access to ak, (ii) no leaks of ak, (iii) immutability of the TCB code, (iv) uninterruptibility of the TCB code, and (v) controlled invocation of the TCB code. In the following, we discuss the implementation of the TCB code and the secure hardware properties on simple and advanced ECUs.

Implementation on Simple ECUs. Simple ECUs implement ALE, our proposed attestation protocol for low-end embedded devices (Chapter 3), to realize the security requirements. Accordingly, simple ECUs are deployed with a protected bootloader $\mathcal{B}$ that is stored in Read-Only Memory (ROM) and contains the TCB code, i.e., the code to execute step (1). The bootloader code is immediately executed when the device starts and makes use of a secure storage to access ak. The secure storage can be implemented, for instance, based on a simple Memory Protection Unit (MPU) [71], an emulated MPU [44], an SRAM PUF [127], or EEPROM that can be hidden [127].

In practice, simple ECUs frequently provide the required secure hardware properties, as they are based on microcontrollers that have shown to exhibit those properties [44, 71, 85, 127]. Furthermore, since several years, all major manufacturers offer ECUs[1] that fulfill the HIS-SHE [46] and/or EVITA [148] standard. Both standards provide a common specification for secure hardware in road vehicles. They specify that ECUs must offer hardware support for immutable and uninterruptible code to implement the secure boot functionality [9]. Recent work [44] has shown that based on a secure boot, a secure storage can be emulated. Therefore, all ECUs that implement the HIS-SHE and EVITA standard also implicitly fulfill the secure hardware requirements to implement ALE (Chapter 3). Note that

1 E.g., AURIX from Infineon; MPC564xB/C, MPC5746M, MPC574xB-C-G from NXP; SPC564B/EC, SPC56ECx from ST; RH850/P1x-C from Renesas.

actual ECUs often offer a real MPU, whose use we prefer over an emulated MPU (e.g., all ECUs mentioned in footnote[1] on the previous page provide an MPU).

Implementation on Advanced ECUs. Advanced ECUs utilize the same measures as simple ECUs to implement the necessary properties. Hence, they also store their boot code and boot data in ROM. Yet, to achieve increased functionality, we require advanced ECUs to also feature the ARM TrustZone technology [6], which offers a privileged second execution environment, called secure world. The secure world is isolated by hardware from the normal world, in which the standard Operating System (OS) and applications are running. Since the ARM Cortex-A15 from 2010, all ARM application processors (Cortex-A) feature the TrustZone technology. Moreover, the new ARMv8-M architecture brings TrustZone to microcontrollers, namely Cortex-M23/M33/M35P processors. Additionally, we require the secure storage of ak to be only accessible from the secure world, which is a common requirement for TrustZone-based security services [49, 120]. To implement the secure storage, manufacturers offer different solutions, such as hardware fuses, TrustZone aware memory controllers, and/or IOMMU [49].

The additional secure hardware features are necessary because advanced ECUs must rely on a different attestation technique than simple ECUs due to their software complexity. Nowadays, a full-featured OS contains thousands of constantly changing files. Predefining which memory regions must be measured to detect malware, as required for simple ECUs, would hence be an impossible task on advanced ECUs. Therefore, we instead present a flexible technique, where any executable code is measured on demand at load-time, before it is executed. Using the additional security features of advanced ECUs, our technique ensures that the measurement code is unmodified, and executed potentially malicious software is unable to tamper with already performed measurements.

More precisely, advanced ECUs are deployed with a bootloader that performs a secure boot [9] of two software systems: (i) a Linux-based OS running in the normal world, and (ii) a software called OP-TEE [99] running in the TrustZone secure world (see Figure 4.2). The secure boot technology ensures that the particular software comes from a trusted party, like the manufacturer, as opposed to an adversary. For this purpose, the software is measured and the measurement is compared with a reference measurement from a stored certificate. In case actual and reference measurement differ or the signature verification of the certificate fails, the software is not executed.

All (automotive) applications running on the advanced ECU are managed by the Linux-based OS, which is deployed with an enabled Integrity Measurement Architecture (IMA) [123]. IMA is part of recent Linux kernels and offers the capability to measure binary files before their execution, including libraries and configuration files. Each measurement is stored in the IMA measurement list and contains, among others, the filename, filepath, and hash value computed over the file.

The second securely booted software, OP-TEE, manages all applications that are running in the TrustZone secure world. In particular, a special Trusted

Application (TA) is deployed in the secure world of advanced ECUs. The TA provides an API that enables the IMA to pass performed measurements to the TA. Passed measurements are concatenated in im ($im = im_1 \| im_2 \| ...$) and stored protected in the secure world. In addition, the TA offers a second API to compute the response key rk. For this purpose, the TA takes a nonce n_B and returns an HMAC rk that is computed with the attestation key ak over n_B and the concatenated measurements im. Note that the computation of rk can also take place on demand in step (2). This is possible, as the computation is handled in the secure world, which prevents potentially malicious code running in the normal world from tampering with the computation.

4.1.3 *Evaluation*

In the following, we first describe our implementation setup. Afterwards, we show and evaluate our measurements.

Implementation Setup

We selected two different hardware boards to represent simple and advanced ECUs, and connected them in a CAN bus and via Ethernet. As a target platform for simple ECUs, we used 5 Olimex ESP32-EVB development boards, which feature 4 MB flash memory, a 240 MHz dualcore 32-bit microprocessor, 100 MBit Ethernet, and a CAN communication module. To implement the required security properties (Section 3.2.1), we locked the bootloader of each ESP32-EVB with the "one-time flash" option, such that it cannot be modified. Furthermore, our deployed bootloader configures the MPU to enable only the bootloader code access to the attestation key ak.

For advanced ECUs and the master ECU, we employed 6 Raspberry Pi 3 Model B+, which feature 1 GB RAM, a 1.4 GHz quadcore ARM Cortex-A53, and Gigabit Ethernet. In addition, we extended each RPI3B+ with a SK Pang PiCAN2 module to enable CAN communication. Unfortunately, we were unable to implement all required security properties on the RPI3B+ (Section 4.1.2), as the RPI3B+ insufficiently protects TrustZone secure world memory and lacks a secure storage. We further acknowledge that our selected hardware lacks certain safety features that can be found in typical automotive hardware, such as a lockstep mode, ECC memory, and a large operating temperature range. However, the implementation of a TrustZone-aware memory controller, secure storage, and automotive safety features only entail a negligible runtime overhead and thus have an insignificant effect on our performance measurements.

Runtime Measurements

Single-Device Runtime. We investigated the runtime overhead to verify a single device. Table 4.1 outlines our averaged runtime measurements for the main

Operation	ESP32-EVB	RPI3B+
Secure Boot Overhead	587.38 ms	508.29 ms
SHA-256 (32 B Flash)	0.04 ms	0.06 ms
SHA-256 (512 kB Flash)	126.71 ms	67.30 ms
SHA-256 (2 MB Flash)	-	281.65 ms
HMAC-SHA-256 (32 B RAM)	0.10 ms	0.08 ms
HMAC-SHA-256 (64 B RAM)	0.12 ms	0.09 ms
HMAC-SHA-256 (1 kB RAM)	0.37 ms	0.32 ms
Ethernet Round Trip Time	0.75 ms	0.44 ms
Ethernet Throughput	49.08 MBit/s	95.70 MBit/s
CAN Round Trip Time	0.17 ms	0.85 ms
CAN Throughput	1.27 MBit/s	1.22 MBit/s

Table 4.1: Averaged measurements on the ESP32-EVB and RPI3B+.

building blocks of our attestation scheme on the ESP32-EVB and the RPI3B+. As shown, the main overhead on both devices comes from the secure boot, which ensures the immutability and uninterruptability of the boot code. The secure boot takes with 587 ms on ESP32-EVB and 508 ms on RPI3B+ much more time than other operations, as it involves multiple computationally expensive signature verifications. Note that on other devices, a secure boot may not be required to implement the secure hardware properties (Section 4.1.2).

Further overhead is imposed by cryptographic operations, which we implemented with the mbed TLS library. During attestation, both devices compute a SHA-256 hash value over all software to be executed. Whereas the ESP32-EVB hashes a 512 kB firmware binary, which takes 127 ms, the RPI3B+ measures 29 files that sum up to a total of 2.5 MB, which takes 326 ms. Moreover, both devices compute two HMACs, which consumes 0.24 ms runtime on the ESP32-EVB and 0.41 ms on the RPI3B+. The RPI3B+ requires more time than the ESP32-EVB because it computes an HMAC over 29 software integrity measurements, as opposed to a single HMAC over the entire firmware binary. Furthermore, the master ECU, an RPI3B+, requires 0.18 ms to recompute the HMACs and verify the attestation response of an ESP32-EVB, and 0.44 ms to verify responses of an RPI3B+.

Additionally, the network communication also entails overhead, as a 16 bytes challenge and 33 bytes response needs to be transmitted during attestation. Interestingly, the performance of the network interfaces is quite different on the ESP32-EVB and RPI3B+. On the ESP32-EVB, CAN is faster than Ethernet and requires only 1.19 ms to transmit both challenge and response. By contrast, on the RPI3B+, CAN is much slower than Ethernet and entails a total communication overhead of 5.95 ms.

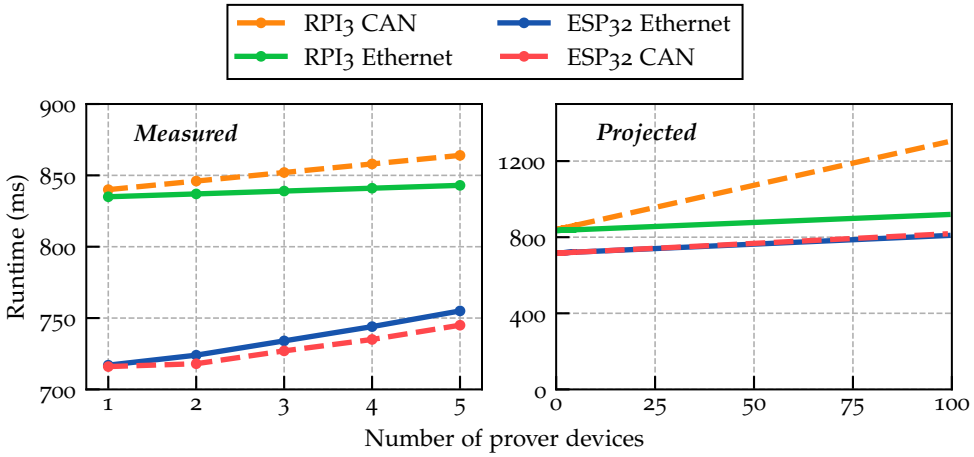


Figure 4.3: Runtime overhead of verifying multiple ECUs in different networks.

Altogether, the runtime overhead to perform the attestation of a single device depends on the device type and network interface. It varies between at least 715 ms (on ESP32-EVB over CAN) and at most 844 ms (on RPI3B+ over CAN).

Multi-Device Runtime. Figure 4.3 depicts the runtime overhead to attest a varying amount of ECUs over CAN and Ethernet. As shown, the fastest attestation is possible with the ESP32-EVB over CAN, while the slowest is the RPI3B+ over CAN. During attestation, safety-critical ECUs can perform many operations in parallel, such as measuring software integrity or computing attestation responses. For this reason, the runtime to attest many devices only slightly increases with a raising number of devices in the network, compared with the overhead to verify a single device. This demonstrates the good scalability of our attestation scheme. In fact, our projections to large networks show that the attestation of up to 100 devices takes even in the worst-case, being a CAN network with only RPI3B+ devices, less than 1.4 seconds. Because a typical driver requires more than 1.4 seconds to get in the vehicle and initiate its start, our scheme entails no perceptible delays for drivers and passengers. Moreover, road vehicles typically contain less than 100 safety-critical ECUs.

4.1.4 *Summary*

We presented an automotive attestation scheme in which a trusted master ECU verifies the software integrity of all safety-critical ECUs in the vehicle each time the vehicle is unlocked and opened. Only after the master has ensured that all safety-critical ECUs are in a trustworthy software state, the master allows the vehicle to move. This way, compromised ECUs are prevented from jeopardizing the safety of the vehicle. To address the heterogeneity of ECUs, we presented two distinct attestation techniques. The first technique targets simple sensor and actuator ECUs, while the second technique is designed for more complex

ECUs, e.g., used for sensor fusion. Since both techniques are applicable to many existing commodity ECUs, our solution entails only minimal deployment costs. We evaluated our scheme in an automotive network that uses CAN and Ethernet. Our results show that the overall timing overhead to verify 100 ECUs is less than 1.4 s. Thus, our scheme causes no perceptible delays for drivers and passengers.

4.2 USE CASE 2: SECURE CODE UPDATES IN MESH NETWORKS

4.2.1 *Motivation and Contribution*

A popular way to connect low-end embedded systems are wireless mesh networks. In mesh networks, all devices cooperate in the distribution of data in the network, forming a decentralized and self-organized network topology. Recent technologies like IEEE 802.11s, IEEE 802.15.4, ZigBee, Z-Wave, or Bluetooth enable embedded systems to establish large wireless mesh networks consisting of numerous embedded systems. Nowadays, these networks are widely used in industrial control, wireless sensor networks, home automation, building automation, military communication, or community networks. To maintain the security of these networks, it is essential that devices provide the capability for remote code updates. In fact, unpatched software vulnerabilities are currently the second most predominant reason for malware infections on embedded devices [76]. Using secure code update mechanisms, the proper installation of software and the erasure of malware can be verified. This enables to reestablish trust in potentially compromised devices by enforcing the proper installation of security patches as well as the erasure of potential malware.

A secure code update scheme for embedded devices in mesh networks must provide several features. First, it has to ensure that devices verify the novelty, integrity, and authenticity of code updates before installation. This feature is necessary to prevent misuse of the code update mechanism, e.g., by downgrading a software or installing malicious code. Second, the scheme must ensure that, appropriately executed, it restores the integrity of the software state on a device, even if the device was compromised before. Thus, an attacker who exploited a vulnerability in the old software to compromise and gain control over a device is removed from the device. However, compromised devices can simply deny the execution of code updates or execute them inappropriately without restoring software integrity. Therefore, after code update execution, the scheme must verify whether all devices are in a trustworthy, i.e., an unmodified and up-to-date, software state. To reduce potential damage caused by compromised devices, the secure code update scheme should exclude untrustworthy devices from the network. Furthermore, the scheme must be scalable, as it should allow for an efficient update of all devices in large mesh networks. Moreover, it should be applicable to already existing commodity low-end embedded devices. In this way, the scheme can be retrofitted to currently deployed systems. Finally, a network

operator issuing a secure code update should be informed about the software integrity of all devices in the network.

However, existing solutions do not satisfy all these requirements. Software- and PoSE-based (Proofs of Secure Erasure) approaches are applicable to commodity devices, but rely on strong security assumptions that are hard to achieve in practice [10, 51, 75, 116, 132]. Additionally, they allow a verifier to perform the attestation with only one device but not a group of devices, as they rely on the assumption that during attestation an adversary is unable to communicate with any other party, except for the verifier. By contrast, hardware-based solutions provide much stronger security guarantees by relying on secure hardware modules. Yet, security architectures which are applicable to low-end embedded systems, such as TyTAN, SMART, TrustLite, or SANCUS, are still in research stage [23, 45, 82, 109]. These architectures have only been implemented as prototypes and their future availability in commodity devices is uncertain.

Contribution. In this section, we present a secure code update scheme for mesh-networked commodity low-end embedded devices that achieves all above mentioned properties. To verify the proper installation of code updates, the scheme relies on ALE, our proposed attestation protocol for low-end embedded devices (Chapter 3). In contrast to existing secure code update solutions (Section 2.1.3), this makes our scheme applicable to many commodity low-end embedded devices as well as to groups of interconnected devices. Receiving a code update, a device checks its freshness and authenticity, and then installs the update. After installation, each device verifies its local software integrity and ensures that only unmodified and up-to-date software runs on the device. To enforce a proper execution of the code update, neighboring devices mutually verify the genuine and up-to-date software state of each other, and establish secure channels only if the verification succeeds. Thus, compromised devices can either refuse an appropriate execution of the code update, whereupon they are excluded from the network, or perform a correct code update, whereby any present malware gets eliminated. Issuing a secure code update for the network, the network operator is able to learn the identity of all trustworthy and all untrustworthy network devices. We implemented the scheme on exemplary low-end embedded systems that were interconnected via ZigBee. Simulation results demonstrate that our scheme scales well and is practically usable in networks with tens of thousands of devices.

Outline. Section 4.2.2 describes our secure code update scheme. In Section 4.2.3, we evaluate the performance of the proposed scheme. Finally, Section 4.2.4 concludes this section.

4.2.2 *Secure Code Update Scheme*

Our secure code update scheme comprises two phases: a deployment phase and an online phase. The deployment phase is executed once, before the initial

initialization of all devices. In the deployment phase, each low-end embedded device $\mathcal{D}_i$ is initialized by the trusted network operator $\mathcal{O}$. After the devices have been deployed, the online phase is executed repeatedly, once for every code update. In the online phase, $\mathcal{O}$ issues a secure code update for all devices in the network.

Deployment Phase

In the deployment phase, the network operator $\mathcal{O}$ generates a unique identifier i and signature key, consisting of a public key pk_i and a private key sk_i, for each device $\mathcal{D}_i$. For the purpose of authenticating devices and for implementing a challenge-based protocol to verify code update installations, we use public-key cryptography. Therefore, the symmetric attestation key ak_i from ALE (see Chapter 3) is now replaced by the private signature key sk_i. Consequently, sk_i is stored in a secure storage that can only be accessed during the execution of the protected bootloader $\mathcal{B}$ (see Section 3.2). Furthermore, $\mathcal{O}$ equips each device $\mathcal{D}_i$ with a device certificate dc_i and a software certificate sc_i, both signed by $\mathcal{O}$ with sk_O ($\mathsf{dc}_i.sig, sc_i.sig$). The device certificate dc_i stores the device class c of $\mathcal{D}_i$, the public key pk_i of $\mathcal{D}_i$, and the identifier i. The software certificate sc_i lists all memory regions on $\mathcal{D}_i$ where the code update routine and the firmware is stored. In addition, sc_i provides hash values over the data of these memory regions ($sc_i.hash$). Thus, sc_i can be used to verify the integrity of the installed software on $\mathcal{D}_i$. In order to indicate the freshness of the software, sc_i also stores a software version number ($sc_i.ver$). Moreover, each device initially stores the public key of the trusted network operator pk_O in $\mathcal{B}$. Both sc_i and dc_i are stored in a mutable and unprotected memory region.

Additionally, $\mathcal{O}$ equips each device with the necessary code to perform a secure code update. Our scheme relies on the untampered execution of code that attests the integrity of the local software state. For this reason, code that implements the attestation routine is stored in $\mathcal{B}$, while the rest of the code (including the actual code update functionality) is stored in a mutable and unprotected memory region (see Figure 4.4). Table 4.2 summarizes relevant definitions used in the deployment and the online phase.

$\mathcal{O}$	Trusted network operator	sk_i	Secret signing key of entity i
$\mathcal{B}$	Protected bootloader	pk_i	Public signing key of entity i
$\mathcal{D}_i$	Device with identity i	dc_i	Device certificate of entity i
sh_i	Secret ECDH key of entity i	sc_i	Software certificate of entity i
ph_i	Public ECDH key of entity i	cu_c	Code update for device class c

Table 4.2: Notation.

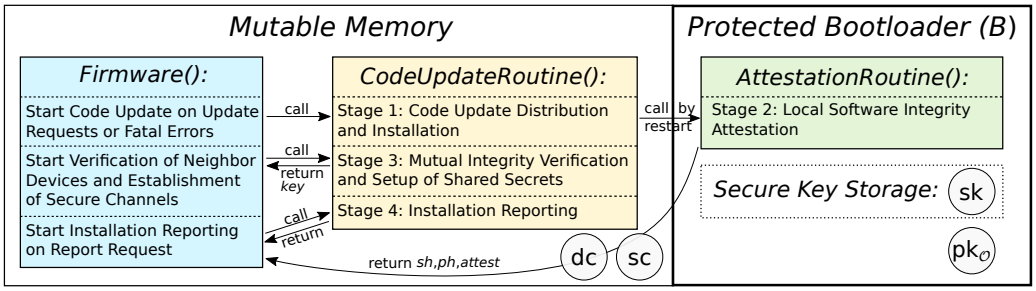


Figure 4.4: Illustration of memory layout and control flow of the online phase.

Online Phase

The online phase consists of four different stages. In the first stage, $\mathcal{O}$ prepares a code update package, which is distributed in the network and installed on the devices. In the second stage, devices invoke the execution of the attestation routine in $\mathcal{B}$. The attestation routine verifies the integrity of the installed software and ensures that a device subsequently executes an unmodified and up-to-date software. Additionally, the attestation routine generates an attest which proves a trustworthy software state by certifying an untampered and complete execution of the attestation routine. In the third stage, neighboring devices verify the software integrity attest of each other. If the verification is successful, devices establish a secure channel. As untrustworthy devices are unable to attest their valid software integrity, they cannot establish communication channels and thus are excluded from the network. In the fourth stage, $\mathcal{O}$ obtains an installation report, which exhibits the software state of all devices in the network. Figure 4.4 shows the memory layout of the code update scheme and illustrates the control flow throughout all stages. In the following, we will explain each stage in detail.

Stage 1: Code Update Distribution and Installation

The online phase starts with $\mathcal{O}$ preparing a code update package *cupkg*. *Cupkg* includes an ascending version number *cupkg.no* and a signature by $\mathcal{O}$, in order to prevent replay attacks and tampering with the code update package. Since devices in the network may be heterogeneous, *cupkg* is able to address multiple device classes. For each device class c in the network, *cupkg* contains a software certificate sc_c, which specifies the correct software configuration for a device of type c. In addition, all software certificates that are contained in *cupkg* store the current *cupkg* version number ($sc.ver = cupkg.no$). Furthermore, for each device class c that should be updated, *cupkg* contains code update data cu_c. Code update data cu_c comprises the binary code of the update as well as instructions for its installation (e.g., addresses where to store the binary code during installation).

After preparing *cupkg*, $\mathcal{O}$ sends a code update request followed by *cupkg* to an arbitrary device in the network. This causes the recipient device to execute stage one in the code update routine (see Figure 4.4). Next, the code update routine receives *cupkg* and stores it in a free memory region. Devices that received

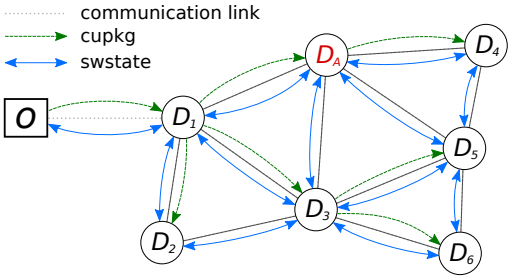


Figure 4.5: *Cupkg* distribution in stage 1 and *swstate* message exchange in stage 3.

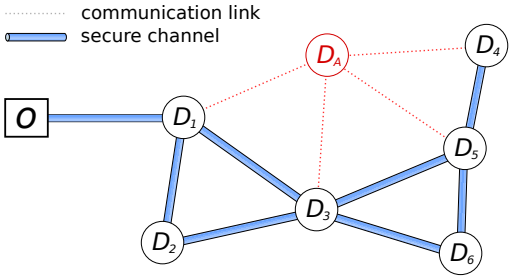


Figure 4.6: Establishment of secure channels in stage 3 in presence of an adversary $\mathcal{D}_A$.

cupkg check whether it contains a valid signature by $\mathcal{O}$ and whether its version number is higher than the last received *cupkg* version number. If both checks pass, devices send a code update request to their immediate neighboring devices and subsequently forward *cupkg* to them. In this way, a flooding propagation of *cupkg* is initiated (see Figure 4.5). Since efficient and secure code disseminations in wireless mesh networks are well-understood [41, 58, 60, 93, 96, 122, 144], we will not elaborate on the distribution of *cupkg*, but instead assume that eventually each device in the network receives *cupkg*.

Devices that received, verified, and forwarded *cupkg* to their neighbors check whether *cupkg* comprises a new code update for their device class. To this end, each device $\mathcal{D}_i$ examines whether *cupkg* contains a cu_k for the local target device class specified in dc_i. If this is the case, a device uses the installation instructions in cu_k to install the update binary code. Note that, since the code update routine is stored in mutable memory, the update routine itself may also be updated during update installation. Furthermore, all devices in the network update their local software certificate to the new software certificate for their device class and then issue an attestation of the local software configuration (see next stage).

We propose that devices also invoke the execution of the code update routine on fatal errors that render devices non-functional. This allows $\mathcal{O}$ to remotely recover devices whose software accidentally became misconfigured or defective.

Stage 2: Local Software Integrity Attestation

In order to attest an untampered and up-to-date software state, devices invoke the execution of the attestation routine. Since the attestation routine is stored in the protected bootloader $\mathcal{B}$ (see Figure 4.4), its execution requires a device reboot. As illustrated in Algorithm 4.1, the attestation routine starts with the retrieval of the protected secret signing key sk. Next, the authenticity of the software certificate *sc* is ensured by verifying whether *sc* was signed by $\mathcal{O}$. If this is the case, *sc* is used to check the local software integrity (denoted by the execution of CheckCodeIntegrity()). Consequently, hash values over all memory regions that are listed in *sc* are taken and compared to the expected reference values specified in *sc.hash*. If all measurements match their reference value, the verification of

the software integrity is successful. Upon a successful verification, the device generates a new Elliptic curve Diffie-Hellman (ECDH) key pair (sh, ph) [95] and computes *attest* by signing *ph* and *sc.ver* with sk. Afterwards, it is ensured that no information about the secret signing key sk gets leaked (denoted by the execution of HideSecret()). As described in Section 3.2, this may involve the erasure of certain memory regions or the execution of specific instructions on some commodity devices. Finally, the firmware is executed and *sh*, *ph*, and *attest* are passed to the firmware (see Figure 4.4). The entry point of the firmware is hardcoded in $\mathcal{B}$. This ensures that the control flow is indeed passed to the firmware, whose integrity was just verified, and not to malicious code that hides somewhere in memory. However, if the verification of the software integrity is unsuccessful, stage one in the code update routine is executed all over again. In this way, devices are able to recover from situations where $\mathcal{O}$ accidentally distributed a defective *cupkg*.

Algorithm 4.1: Execution of AttestationRoutine() (located in $\mathcal{B}$).

```
1:  procedure AttestationRoutine(sc)
2:      sk ← RetrieveSecret()
3:      if Verify(pk_O; sc.sig; sc.content) and CheckCodeIntegrity(sc.hash) :
4:          sh, ph ← GenKey()
5:          attest ← GenSig(sk; ph||sc.ver)
6:          HideSecret(sk)
7:          StartFirmware(sh, ph, attest)
8:      else :
9:          HideSecret(sk)
10:         StartCodeUpdateRoutine()
```

We would like to point out that a valid *attest* proves that $\mathcal{D}$ runs a firmware and code update routine whose integrity was successfully verified using a software certificate with the version *sc.ver*. One reason for this are the three device properties (see Section 3.2). They prevent an adversary from tampering with the attestation routine, accessing sk outside of the attestation routine, and interrupting the execution of the attestation routine. Another reason is the design of the attestation routine, which prevents an adversary from generating a valid *attest* without executing the attestation routine from the beginning. This is due to the first instructions of the attestation routine, in which sk is retrieved and thus must initially be executed to sign *attest* correctly. However, executing the attestation routine from the beginning leads to its execution in entirety (see third device property). This inevitably executes code which ensures that no information about sk gets leaked, that the software integrity of $\mathcal{D}$ conforms to *sc*, and that the firmware, and no unverified code, gets executed next. Tampering with the input of the attestation routine is also unpromising for the adversary. The only mutable data that the attestation routine relies on is *sc*. However, *sc*'s integrity is verified before it is used to check the local software integrity. Using

old *sc*s as input for the attestation routine, devices can pass the local software integrity verification with an outdated software state. Nevertheless, as we will see in the next stage, this will be detected during the verification of *attest* by neighboring nodes.

Stage 3: Mutual Integrity Verification and Setup of Shared Secrets

In the third stage, each device reaches out to neighboring devices in the network, i.e., devices within immediate communication range. If a device $\mathcal{D}_i$ finds a neighbor $\mathcal{D}_n$ whose software state has not yet been verified, it invokes a mutual verification. For this purpose, $\mathcal{D}_i$ generates a $swstate_i$ message comprising $attest_i$, ph_i, and dc_i, and sends this message to $\mathcal{D}_n$. Upon receiving $swstate_i$, $\mathcal{D}_n$ generates a $swstate_n$ message and sends it to $\mathcal{D}_i$ (see Figure 4.5). Next, both devices invoke the execution of stage three in their code update routines (see Figure 4.4) to verify each others' software integrity and establish a shared secret. Algorithm 4.2 illustrates this process in pseudocode.

Algorithm 4.2: Software integrity verification of a neighbor device $\mathcal{D}_n$ on $\mathcal{D}_i$.

```
1: procedure VerifyNeighborSoftwareIntegrity(swstate_n)
2:     attest_n, dc_n, ph_n := swstate_n
3:     key ← ⊥
4:     if   VerSig(pk_O; dc_n.sig; dc_n.content)
5:     and  VerSig(dc_n.pk_n; attest_n; ph_n||sc_i.ver) :
6:         key ←KeyExchange(sh_i, ph_n)
7:     return key
```

In order to verify the software state of $\mathcal{D}_n$, $\mathcal{D}_i$ initially checks dc_n using pk_O. Next, $\mathcal{D}_i$ verifies whether $attest_n$ corresponds to the received ph_n and the latest software version, which $\mathcal{D}_i$ stores in its local software certificate ($sc_i.ver$). A successful verification ensures that $\mathcal{D}_n$ is in a software state that corresponds to a *sc* from $\mathcal{O}$'s latest *cupkg*. Thus, it ensures the integrity as well as the up-to-dateness of $\mathcal{D}_n$'s software state. In addition, verifying $attest_n$ confirms the integrity and the authenticity of ph_n. If the verification of dc_n and $attest_n$ is successful, $\mathcal{D}_i$ uses its own secret ECDH key sh_i and $\mathcal{D}_n$'s public ECDH key ph_n to perform a key exchange and establish a shared secret *key*. Note that if $\mathcal{D}_n$'s verification of $\mathcal{D}_i$'s software state is also successful, both parties agree on the same *key*. However, if any of the verifications fail, $\mathcal{D}_i$ regards the software state of $\mathcal{D}_n$ as untrustworthy and does not reconstruct a shared secret. Next, the attestation routine returns and passes *key* to the firmware (see Figure 4.4). If the verification failed, the firmware causes $\mathcal{D}_i$ to send $\mathcal{D}_n$ a message that indicates a failure. Nevertheless, $\mathcal{D}_n$ can re-request a mutual integrity verification with $\mathcal{D}_i$ to recover from connection breaks or other repairable errors.

If the verification was successful on both sides, $\mathcal{D}_i$ and $\mathcal{D}_n$ use *key* to establish a confidential and authenticated channel. This channel is used for any further

communication between both parties. In this way, devices whose software is in an untrustworthy state are effectively excluded from communication. An adversary may try to pass the mutual software state verification by replaying a *swstate* messages recorded from a trustworthy device. However, in doing so, the adversary is not in the possession of the *sh* that correspond to the replayed *swstate* message. For this reason, the adversary is not able to reconstruct the correct *key* and the attack will be detected during the establishment of the secure channel. Figure 4.6 illustrates a scenario in which a compromised device $\mathcal{D}_A$ is unable to attest its software state towards its neighboring devices and thus is unable to establish a communication channel with them.

Stage 4: Installation Reporting

The fourth stage starts with $\mathcal{O}$ requesting an installation report from the network. For this purpose, $\mathcal{O}$ initially uses the approach explained in stage three to establish a secure channel with an arbitrary trustworthy device $\mathcal{D}_i$ in the network. In order to pass the integrity verification by $\mathcal{D}_i$, $\mathcal{O}$ generates a signature key pair, issues a dc that authenticates the generated key, and uses the key to compute *attest*. Next, $\mathcal{O}$ sends $\mathcal{D}_i$ a request for an installation report over the established channel. Devices that receive a report request invoke the execution of stage four in their code update routines (see Figure 4.4). The report request is used to construct a spanning tree whose root is $\mathcal{O}$. For this purpose, $\mathcal{D}_i$ broadcasts the request over secure channels to all trustworthy neighboring devices, which in turn broadcast the request. Broadcasting is repeated until the report request reaches leaf nodes in the spanning tree, i.e., nodes whose neighbors all have received the request. Leaf nodes then generate an installation report, which initially contains the identifier of the particular leaf node. Afterwards, the installation report incrementally gets propagated back to the root of the spanning tree. At each hop, a node aggregates the report from its child nodes, includes its own identifier, and then forwards the aggregated report to its parent node. Above a certain number of aggregated identifiers, it is useful to encode the report as an n-bit array, where a flipped bit at position k indicates that $\mathcal{D}_k$ is in a trustworthy state. Eventually, the installation report gets transmitted from $\mathcal{D}_i$ to $\mathcal{O}$. Since $\mathcal{O}$ knows the identifiers of all deployed devices, $\mathcal{O}$ can also assess the precise identifiers of all untrustworthy devices. This may serve as a first step towards physically locating and recovering compromised devices.

Note that listing the precise identifiers of all devices in the network causes a considerable communication overhead in large mesh networks with many devices. If $\mathcal{O}$ does not require detailed information about the identity of trustworthy and untrustworthy devices, it is reasonable to implement a more coarse-grained report type. For instance, $\mathcal{O}$ could initially only request for the total number of trustworthy devices or the number of trustworthy devices per device class.

4.2.3 *Evaluation*

Setup

We implemented the proposed secure code update scheme on Stellaris EK-LM4F120XL microcontrollers. The Stellaris is a low-cost embedded system from Texas Instrument which features an 80 MHz ARM Cortex-M4F microprocessor and provides 256 kB of protectable flash memory. To enable wireless mesh connectivity based on the ZigBee standard, we equipped the Stellaris microcontrollers with CC2530 BoosterPacks from Anaren. In the following, we consider a homogeneous network of Stellaris microcontrollers in a static network topology. We measured network and computational delays of our implementation in small real word mesh networks. In order to evaluate the scalability of the secure code update scheme, we simulated large-scale networks based on our measurements. We found out that the network topology plays an important role for the code update runtime. This is due to the high communication costs for the transmission of the binary code updates.

We implemented the key exchange using Elliptic Curve Diffie-Hellman (ECDH) with Curve25519 [18]. For the signature scheme, we used an Edwards-curve Digital Signature Algorithm (EdDSA) called Ed25519, which is based on Curve25519 [19]. We implemented the hash function using SHA-512, while the secure and authentic channel uses AES in Galois/Counter Mode (AES-GCM).

Runtime Measurements

Network Runtime Performance. For unicast messages between two neighboring nodes in the mesh network, we measured an average maximum throughput of 35.0 kbps on the application layer. Although the measured throughput is only a fraction of the theoretical maximum throughput of 250 kbps in ZigBee networks, other performance evaluations revealed similar performance losses in reality [25]. In addition, we measured an average one-hop round-trip time (RTT) of 13.5 ms with the smallest message size (1 byte) and 18.5 ms with the biggest message size (81 bytes). Carrying out measurements for the broadcast throughput, we found out that the broadcast frequency on the CC2530 BoosterPack is limited according to the specification of the ZigBee Pro stack. In practice, we measured a maximum broadcast throughput of 0.65 kbps. Thus, for performance reasons, we implemented all network communication as unicast transmissions. During protocol execution, additional overhead is generated through the restart of the devices. We measured that a full device restart, comprising an initialization of the device and a join in the mesh network, takes on average 2338 ms.

Cryptographic Runtime Performance. Table 4.3 shows an excerpt of our cryptographic runtime measurements on the Stellaris microcontroller. We would like to stress that we based our implementation on platform independent and unoptimized C code [17, 90]. Recent works have shown that assembler optimized

Algorithm	Function	Runtime	Function	Runtime
ed25519	GenKey()	18 ms	KeyExchange()	48 ms
	GenSig(16 B)	19 ms	VerSig(16 B)	51 ms
	GenSig(1024 B)	22 ms	VerSig(1024 B)	53 ms
AES-GCM	AuthEncrypt(16 B)	0.1 ms	AuthDecrypt(16 B)	0.1 ms
	AuthEncrypt(1024 B)	1.8 ms	AuthDecrypt(1024 B)	1.8 ms
SHA-512	Hash(16 B)	0.4 ms	Hash(20 480 B)	54.7 ms
	Hash(1024 B)	3.1 ms	Hash(163 840 B)	435.1 ms

Table 4.3: Cryptographic runtime measurements on the Stellaris.

code for low-end embedded systems can improve the performance of cryptographic operations by orders of magnitudes [37, 42]. We presume that similar performance improvements are also possible on the Stellaris platform, if the used cryptographic primitives were optimized for ARM Cortex-M4F microprocessors.

Storage Consumption

Compared to a naïve code update approach that only distributes the binary code of the update but provides no security, our scheme requires additional storage for data. In fact, each device must store sc (ca. 212 bytes), dc (100 bytes), pk_O (32 bytes), sk (64 bytes), ECDH keys (96 bytes), and shared secrets (32 bytes per neighbor). Hence, with k being the number of neighboring devices, the storage overhead for data adds up to $504 + 32 \cdot k$ bytes.

Further storage consumption arises due to the size of the code. Our reference implementation, which we use throughout this performance evaluation, requires 66 kB of protected storage in $\mathcal{B}$. However, almost all the storage is spent for the implementation of the Ed25519 signature scheme. By using an Ed25519 implementation that is particularly suited for low-memory systems [15], we were able to reduce the size of $\mathcal{B}$ to 7.7 kB, albeit increasing the runtime for cryptographic operations[2]. This smaller implementation makes our scheme applicable to all commodity low-end embedded devices analyzed in Section 3.2.2, since all of them offer at least 8 kB of protectable flash memory. Reusing the signature scheme in $\mathcal{B}$, additional 15.1 kB of code in mutable memory are consumed for the implementation of the network communication, key exchange, encryption and decryption, and protocol logic. In total, our reference implementation consumes 81.1 kB and our code size optimized implementation consumes 22.9 kB of storage. This is an acceptable overhead of 31.6 % and 8.9 % of the available storage on the Stellaris platform.

2 Yet, existing works have shown that a signature scheme which achieves about the same runtime performance than our reference implementation can be implemented in less than 4 kB of code by using platform dependent assembler directives [37, 103].

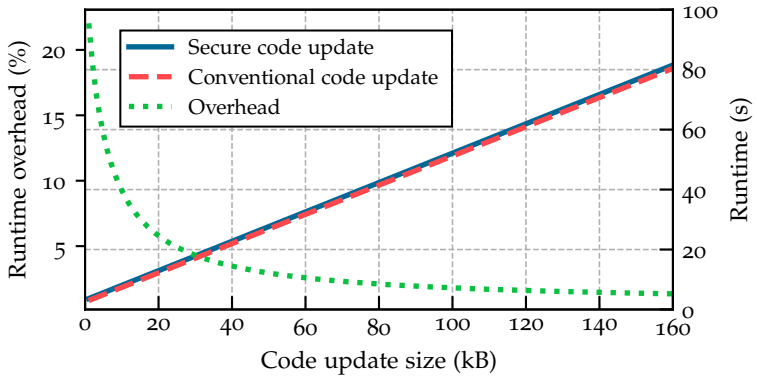


Figure 4.7: Single device runtime performance with varying code update sizes.

Single Device Secure Code Update Runtime

We simulated the runtime of our code update scheme under various conditions and compared it to a conventional code update approach. The conventional code update approach distributes and installs code updates and ensures the authenticity, integrity, and freshness of updates on the devices. However, it does not exclude devices that are in an untrustworthy software state from the network and also provides no report listing all trustworthy devices for the network operator.

Figure 4.7 compares the runtime on a single device between our secure code update approach and the conventional code update approach with varying code update sizes. It shows the runtime of both approaches in seconds as well as the percentage overhead of our secure approach. The mesh network consists of 1024 nodes which are arranged in a binary tree topology. Since devices in the network require different amounts of time to perform the code update, e.g., some devices transmit a smaller installation report or they need not forward the new code update to neighboring devices, we averaged the runtime over all devices in the network. Figure 4.7 illustrates that the size of the code update has a linear impact on the code update runtime. This is almost entirely due to the transmission time of the code update. In fact, the runtime overhead of our secure approach is nearly independent of the code update size, as it increases only slightly from 0.7 seconds with a 1 kB code update to 1.1 seconds with a 160 kB code update. For that reason, compared with the conventional scheme, the runtime overhead decreases from 22.0% to 1.4% with an increasing size of the code update.

Figure 4.8 shows the runtime for a 30 kB code update on a single device with a varying number of neighbor devices. We distributed the code update to the measured device first, which is why all surrounding neighbor devices are supplied with the code update during protocol execution. This causes a linear increase of the code update runtime. However, the additional runtime of our secure update approach also increases linearly with the number of neighbor devices. This is due to the time neighboring devices require to mutually verify

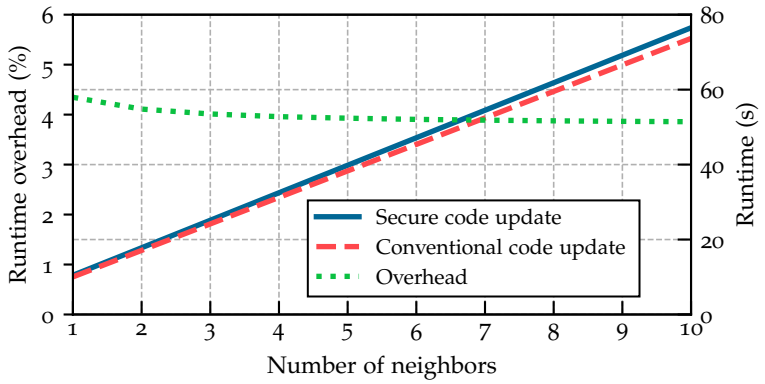


Figure 4.8: Single device runtime performance with a varying number of neighbors.

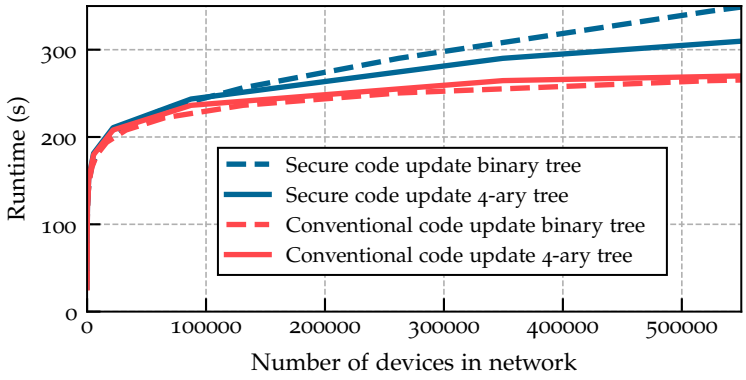


Figure 4.9: Network runtime performance with varying network sizes.

each others' software state during protocol execution. Thus, with a varying number of neighbor devices, the runtime overhead remains rather constant at approximately 4%.

Network Secure Code Update Runtime

We further evaluated the total runtime required to perform a secure code update with all nodes in the mesh network. Figure 4.9 shows the total runtime for a 30 kB code update with varying numbers of nodes, using the secure or the conventional code update approach. The network topology is arranged as a binary tree or a 4-ary tree. The figure demonstrates that due to the tree network topologies, the code update runtime increases logarithmically with the number of devices in the network. We configured our secure update scheme to report the precise device ids of all trustworthy devices to the network operator. As this causes the installation report to grow proportional with the network size, the gap between the runtime of our secure approach and the conventional approach increases considerably when the network contains more than 100.000 devices. In such large network, our secure code update scheme performs better if the network is arranged in a broader but flatter network topology, as this decreases

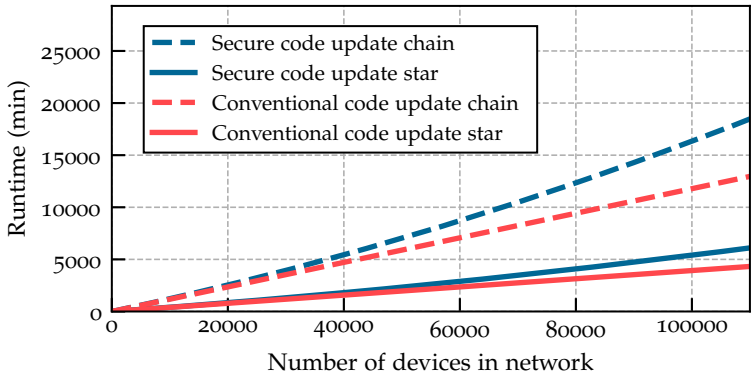


Figure 4.10: Network runtime performance with varying network sizes.

the average size of the report (among others, it increases the number of leaf nodes that must only transmit their own id to the parent node). Therefore, in networks with more than 106.000 devices, our code update scheme performs better in a 4-ary tree network topology than in a binary tree topology. Nevertheless, for smaller networks, the runtime overhead is quite low in tree network topologies. To be precise, the runtime overhead remains below 2% for up to 25.000 devices and is less than 5% for up to 100.000 devices.

However, mesh networks could also embrace unfavorable topologies. Figure 4.10 depicts the total runtime performance for a 30 kB secure code update in a network with a chain topology and a star topology. The star topology is constituted of three device chains branching off a central star device. Figure 4.10 shows that in such an inconvenient network topology, the runtime for a code update attains extremely high values. This is caused by the long transmission time for the code update and the installation report. Nevertheless, even in the worst case, which is the chain topology, the overhead of our secure approach compared to the conventional approach is below 2% for up to 4.000 devices and less than 11% for up to 30.000 devices in the network.

We would like to stress that the overhead is largely introduced by transmitting the precise ids of trustworthy devices to the network operator. If we instead configure our scheme to report only the total number of trustworthy devices to the operator, the network runtime overhead becomes almost negligible compared to the conventional approach. In fact, this way, the runtime overhead is less than 1.5% for 10 devices and less than 0.35% for networks with 500.000 devices.

4.2.4 *Summary*

We presented a secure code update scheme for commodity low-end embedded devices that are connected in large mesh networks. Our scheme offers desirable security features for patching software vulnerabilities in these networks. It enforces that all devices which properly execute the code update only run unmodified and up-to-date software. Devices that refuse a proper execution of

our scheme, and thus run outdated or compromised software, are detected by their neighboring devices and excluded from the network. Issuing a secure code update, the network operator learns which devices are in a trustworthy and which devices are in an untrustworthy software state. Since our scheme is based on ALE (Chapter 3), it can be applied to a broad range of existing commodity low-end embedded devices. In addition, we showed that the scheme scales well and is practically usable in networks with tens of thousands of devices. Compared to a conventional code update, which offers none of the described security features, our scheme imposes a runtime overhead of 2.1% in the best case and 11.9% in the worst case for a network with 30.000 devices and a firmware update size of 30 kB. Thus, our solution is also well suited for future developments, where we expect networks with low-end embedded devices to increase in size.

5

ATTESTATION OF HIGHLY DYNAMIC AND DISRUPTIVE NETWORKS

Collective attestation protocols aim at an efficient attestation of many, or even all, devices in the network. To this end, collective attestation protocols typically arrange a virtual spanning tree during attestation, which enables the efficient transmission and aggregation of information between prover devices and the network operator. However, maintaining a spanning tree topology is inefficient or even inapplicable when devices in the network are mobile or lack continuous connectivity. In this chapter, we present SALAD, a collective attestation protocol for highly dynamic and disruptive networks. SALAD uses a novel distributed approach, where devices incrementally establish a common view on the integrity of all devices in the network. In contrast to other attestation protocols, SALAD performs well in highly dynamic and disruptive network topologies, increases resilience against targeted Denial of Service (DoS) attacks, and allows the network operator to obtain the attestation result from any device. Moreover, SALAD is capable of mitigating physical attacks in an efficient manner, which is achieved by adapting and extending recently proposed aggregation schemes. We demonstrate the security of SALAD and show its effectiveness by providing large-scale simulation results.

Remark. Parts of this chapter have been published in [83].

5.1 MOTIVATION AND CONTRIBUTION

Collective attestation protocols (Section 2.1.2) aim at the efficient attestation of many interconnected devices. To achieve scalability, collective attestation protocols typically rely on a virtual spanning tree, whose root is the verifier. During protocol execution, devices verify their child devices in the spanning tree and propagate an aggregated report to their parent devices. In this way, aggregated reports are incrementally propagated back to the verifier, who eventually receives a report that indicates all healthy network devices. Note that this approach is applied in our secure code update scheme (Section 4.2) as well as in our subsequently proposed protocols SCAPI (Chapter 6) and PASTA (Chapter 7), albeit in a modified form.

A tree-based attestation is very efficient in fully-connected and quasi-static networks. However, in various use cases, device groups operate in sparsely populated networks with high device mobility, e.g., in environmental monitoring [128], offshore exploration [57], agricultural production [26], or emergency communication [143]. In these highly dynamic and/or disruptive networks, tree-based attestation approaches are inefficient or even inapplicable. This is because high

network dynamics lead to reallocations in the spanning tree, so that devices that are neighbors in the spanning tree may shortly afterwards already be several hops away from each other. In addition, network disruptions may break such distant routes over several hops after a short time, such that devices are even unable to communicate with their (virtual) neighbors in the spanning tree in the first place. Furthermore, in constantly partitioned networks, only a fraction of all devices may take part in the construction phase of the spanning tree and it is unclear how all other devices participate in the attestation.

Contributions. In this chapter, we present SALAD, a collective attestation protocol for highly dynamic and disruptive networks. Instead of routing attestation results along a (virtual) spanning tree [7, 11, 28, 68], SALAD pursues a distributed approach where all devices uniformly establish the attestation result. For this purpose, devices within communication range mutually attest the software integrity of each other. Upon success, devices exchange their accumulated attestation proofs, which demonstrate the integrity of further devices in the network. In this way, devices gradually expand their view on the software integrity of other devices in the network, until all devices eventually share the same result about the network state.

The distributed approach of SALAD brings several benefits in dynamic and disruptive networks. First, it increases resilience against targeted Denial of Service (DoS) attacks, in which the communication of only a few devices is impeded to prevent an attestation of many other devices. Second, the verifier can obtain the attestation result from any network device, hence, does not require to hold a connection to a specific, potentially moving, device. Finally, as attestation proofs are shared among all devices using simple device-to-device communication, no routing mechanism is required. This not only saves communication and computational resources, but, most importantly, enables a seamless operation in highly dynamic and/or disruptive networks.

Besides detecting software attacks, SALAD also considers physical attacks. Attestation protocols that detect physical attacks, e.g., DARPA [68], US-AID [65], SCAPI (Chapter 6), or PASTA (Chapter 7), rely on assumptions (Section 2.4) that render them inapplicable in highly disruptive network topologies. For this reason, SALAD only focuses on the mitigation of physical attacks, but not on their detection. In SALAD, each device authenticates its attestation report with a unique key. Thus, a physical adversary can only forge a valid attestation result for devices that have been physically tampered with, but not for other devices. Unfortunately, existing techniques to aggregate, i.e., compress, the emerging additional authentication tags [7, 28, 68] can only be applied in tree-based protocols. This is because they are unable to aggregate intersecting attestation reports[1], which frequently occur in SALAD's distributed attestation approach. However, small attestation reports are vital in dynamic networks. They not only decrease communication, runtime, and energy consumption, but also ensure that

1 E.g., existing aggregation schemes are unable to merge an aggregate containing device A and B with an aggregate containing device B and C, as both aggregates contain B.

devices are able to exchange their reports in the first place, since communication links can break any time. Therefore, we propose two extensions to existing aggregation techniques that make them applicable in SALAD. Compared with a basic solution, the extended schemes are able to decrease the communication by up to 40% and runtime by 95%. To further decrease the overhead, we present an approach that enables the parameterization of security and performance in the proposed aggregation schemes. We will show that using this tradeoff, communication can be decreased by more than 90% and runtime by 98% while providing sufficient statistical security for common attestation use cases.

Moreover, we demonstrate the security and performance of SALAD. Although our protocol is suitable for many device types and network topologies, we will specifically show its applicability in challenging network scenarios, namely, ZigBee connected low-end embedded devices moving with high (drone) speed in large areas. To this end, we provide simulation results based on measurements of SALAD on low-end embedded devices.

Outline. In Section 5.2 we present SALAD, our attestation protocol for highly dynamic and disruptive networks. Section 5.3 describes different aggregation schemes that enable SALAD to efficiently mitigate physical attacks. In Section 5.4, we evaluate the performance of SALAD. Section 5.5 concludes this chapter.

5.2 SALAD: SECURE AND LIGHTWEIGHT ATTESTATION OF DYNAMIC NETWORKS

Our attestation protocol, named SALAD, consists of two different phases. The *deployment phase* (Section 5.2.1) is executed once, in a trusted environment by the network operator $\mathcal{O}$. In this phase, all devices are initialized and deployed. Afterwards, $\mathcal{O}$ can repeatedly execute the *attestation phase* (Section 5.2.2) to verify the software integrity of all devices in the network.

5.2.1 *Deployment Phase*

In the deployment phase, the network operator $\mathcal{O}$ initializes the Trusted Execution Environment (TEE) of each device $\mathcal{D}_i$ with a unique device identifier i, $\mathcal{O}$'s public key $\mathsf{pk}_{\mathcal{O}}$, and a unique attestation phase identifier $curpid_i$. We implement *curpid* as a counter that is initially set to zero.

In addition, each $\mathcal{D}_i$ stores a symmetric attestation key ak_i, required to generate an attestation report that attests the software integrity of $\mathcal{D}_i$ towards $\mathcal{O}$. To reduce the size of attestation reports, SALAD makes use of specific aggregation schemes, which are described in the next section (Section 5.3). Procedures that involve our aggregation schemes are the generation, partitioning, aggregation, and verification of attestation reports, denoted by GenRep(), SliceRep(), AggRep(), and VerRep(), which are used as black boxes in this section.

Acronym	*Usage*
$\mathcal{O}$	trusted network operator
i	unique identifier for device $\mathcal{D}_i$
$\mathsf{sk}_{\mathcal{O}}, \mathsf{pk}_{\mathcal{O}}$	signature key pair of $\mathcal{O}$
ck_{ij}	channel key between $\mathcal{D}_i$ and $\mathcal{D}_j$
$\mathsf{sk}_i, \mathsf{pk}_i$	$\mathcal{D}_i$'s asymmetric key pair for KeyExchange()
ak_i	$\mathcal{D}_i$'s attestation key for GenRep()
$cert_i$	$\mathcal{D}_i$'s certificate to authenticate pk_i
KDF($string$)	derives key from secret
GenSig(key; msg)	generates signature
VerSig(key; sig; msg)	verifies signature
GenMac(key; msg)	generates message authentication code
VerMac(key; mac; msg)	verifies message authentication code
KeyExchange(sk; pk)	establishes shared secret via ECDH
VerSW(vss)	checks software state according to vss
GenRep(key; i; sp)	generates attestation report
SliceRep($desc$; rep)	slices local report for receiver device
AggRep(rep_1; rep_2)	aggregates two attestation reports
VerRep($k_1 \ldots k_n, rep$)	verifies attestation report

Table 5.1: Overview of our notation.

Furthermore, devices implement a function VerSW(vss) that takes vss, a set of valid software states, as input. VerSW(vss) measures the local software state (vss may describe what should be measured) and compares the measurement to the expected measurement in vss. If both values match, the device is assumed to be in a healthy software state and VerSW() returns *true*. Otherwise VerSW() returns *false*. We deliberately abstracted from any implementation details of VerSW() to support a wide range of integrity measurement mechanisms. In practice, VerSW() may be implemented as a simple hash function measuring the binary code of the software, or as an advanced control-flow-attestation mechanism that also protects against runtime attacks [3].

We henceforth assume that any two devices $\mathcal{D}_i$ and $\mathcal{D}_j$ in the network share a common channelkey ck_{ij}. For performance reasons, pairwise channel keys can be preinstalled, if devices provide enough free storage. Otherwise, each device $\mathcal{D}_i$ holds an asymmetric key pair $(\mathsf{sk}_i, \mathsf{pk}_i)$ as well as a certificate $cert_i$, containing a signature over pk_i from $\mathcal{O}$. This enables devices to establish an authenticated channelkey on demand. To this end, both devices exchange their *cert* and pk, verify the other's pk and *cert* with $\mathsf{pk}_{\mathcal{O}}$, and then perform a key exchange, e.g., Elliptic Curve Diffie Hellman (ECDH), using their own sk and the other's pk.

Furthermore, $\mathcal{O}$ equips all devices with the functionality to perform the attestation phase (Section 5.2.2). Except for network message processing, any protocol code is stored and executed inside the TEE of devices. For convenience, we assume that all devices are initialized at once. Yet, $\mathcal{O}$ may deploy a new device at any time by initializing the new device as described. Table 5.1 summarizes relevant definitions.

5.2.2 *Attestation Phase*

Overview. The attestation phase allows the network operator $\mathcal{O}$ to verify the software integrity of all devices in the network. It consists of four steps: (1) $\mathcal{O}$ prepares an attestation initiation and sends it to an arbitrary device in the network. (2) Devices that receive the attestation initiation verify and propagate it to their neighboring devices, i.e., all devices that are within direct communication range. Furthermore, they generate their own attestation report, which attests the software state of the particular device. (3) Neighboring devices mutually exchange attestation reports. After ensuring the software integrity of a neighboring device, a device aggregates its stored attestation report with the neighbor's report. This way, each device holds an aggregated report that gradually contains more and more healthy devices. (4) $\mathcal{O}$ reconnects to the network, receives the attestation report from an arbitrary device, and verifies it. If the report is valid, $\mathcal{O}$ learns the identity of participating network devices with a healthy software state. Finally, $\mathcal{O}$ either stops the attestation phase or waits for more devices to participate in the attestation. In the following, each step of the attestation protocol, formalized in Figure 5.1, is described in detail.

(1) Attestation Initiation. First, $\mathcal{O}$ defines vss, the set of valid software states, which describes all software configurations that $\mathcal{O}$ regards as healthy, e.g., because they represent the correct and most recent software configurations. After setting vss, $\mathcal{O}$ increases the phase id pid, which uniquely identifies the current attestation phase, and chooses appropriate security parameters sp for the attestation report. Details about sp are given in the next section (Section 5.3). Moreover, $\mathcal{O}$ computes a signature sig over vss, pid and sp using $\mathsf{sk}_{\mathcal{O}}$. Finally, $\mathcal{O}$ connects to an arbitrary device $\mathcal{D}_i$ and emits an initiation message msg_{init}, containing vss, pid, sp, and sig.

(2) Generation of Attestation Reports. A device $\mathcal{D}_i$ that receives an attestation initiation msg_{init} verifies its freshness and authenticity by checking whether the phase identifier is new and the signature from $\mathcal{O}$ is valid. In addition, $\mathcal{D}_i$ uses the specified valid software states vss to verify whether it is in a healthy (VerSW() returns true) or compromised software state. Only if all checks pass, $\mathcal{D}_i$ will update its current phase identifier $curpid_i$ to pid and generate an attestation report rep_i (GenRep()), which proves that $\mathcal{D}_i$ is in a known good state according to vss. Generating an attestation report involves the use of one of the aggregation schemes presented in Section 5.3. For now, it is only important to note that

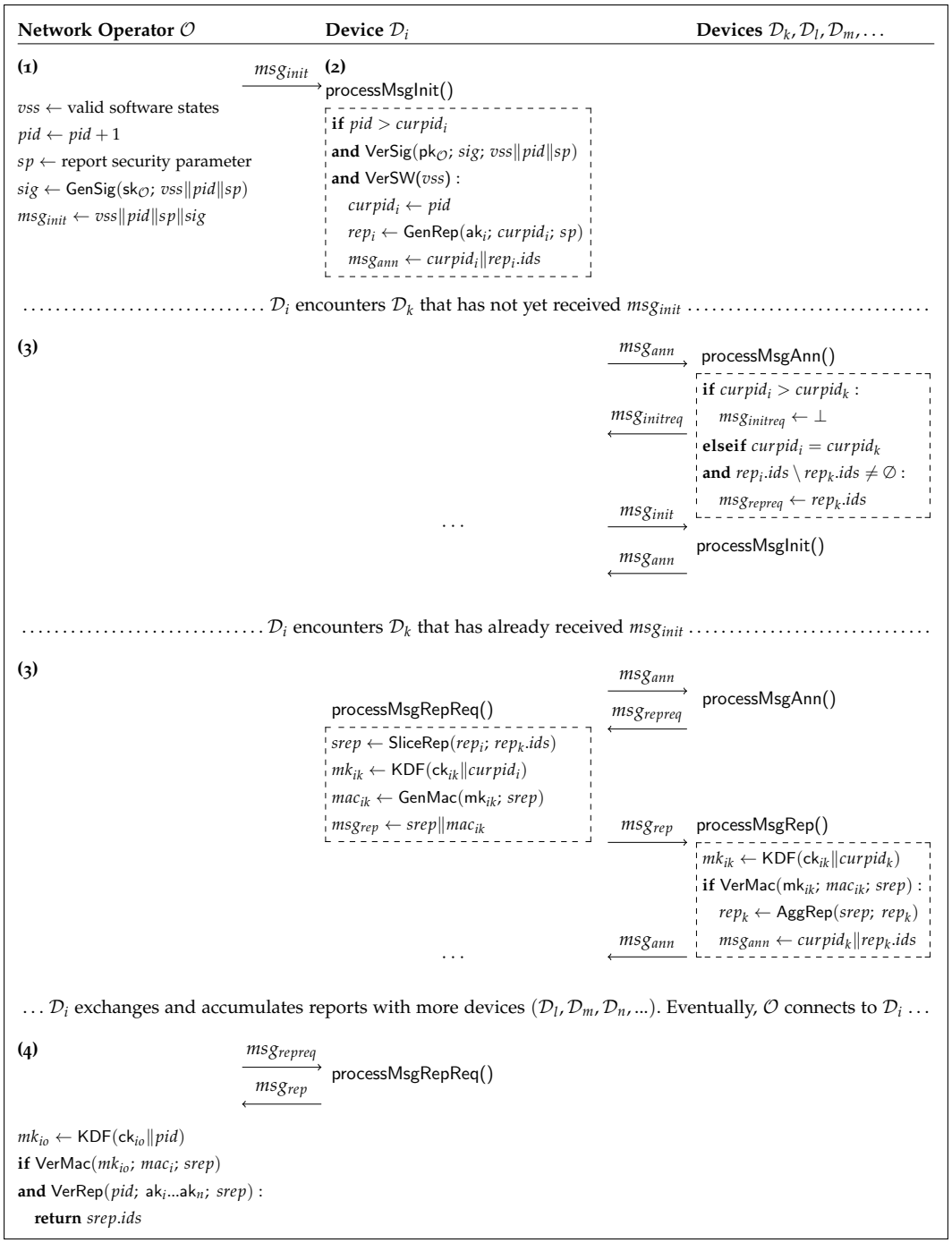


Figure 5.1: The attestation protocol after secure channel establishment, i.e., all devices share a channel key ck and know their identities.

each report *rep* consists of two parts: (i) a device description $rep.ids$, listing the identifier of all healthy devices that contributed to *rep*, and (ii) a data structure $rep.proofs$, which provides the actual proof that all listed devices in $rep.ids$ are healthy. Initially, $\mathcal{D}_i$'s device description $rep_i.ids$ contains the identifier i and $rep_i.proofs$ consists of a MAC computed over the attestation phase id $curpid_i$ with $\mathcal{D}_i$'s attestation key ak_i. Finally, $\mathcal{D}_i$ generates an announcement message msg_{ann}, which is used to announce towards neighboring devices that $\mathcal{D}_i$ holds a new attestation report. Message msg_{ann} contains the phase identifier $curpid_i$ as well as the currently stored device identifiers in the report $rep_i.ids$, which is initially only $\mathcal{D}_i$'s identifier i.

We would like to emphasize that all protocol code is executed inside the TEE of devices (Section 5.2.1). Due to the assumption on the TEE (Section 2.4), Adv_{sw} is unable to generate a valid report *rep* and cannot manipulate the valid software states *vss* as well as the phase id *curpid*. Furthermore, as each attestation proof is authenticated with a unique attestation key, Adv_{hw} can only forge attestation reports for physically compromised devices, where Adv_{hw} has broken the TEE and gained access to the attestation keys. Yet, for all other devices, Adv_{hw} is unable to forge a valid attestation proof.

(3) Distributed Report Aggregation. After completing Step 2, a device $\mathcal{D}_i$ broadcasts msg_{ann} to announce its attestation report to other devices $\mathcal{D}_i$ encounters. A device $\mathcal{D}_k$ that receives msg_{ann}, checks the freshness and relevance of msg_{ann}, which is indicated by the phase id *pid*. If $\mathcal{D}_k$ has not yet received the issued attestation initiation, which is only the case if its stored phase id $curpid_k$ is smaller than the just received $curpid_i$, then $\mathcal{D}_k$ will ask $\mathcal{D}_i$ through a message $msg_{initreq}$ to forward the initial attestation message msg_{init} from $\mathcal{O}$. On receiving msg_{init}, $\mathcal{D}_k$ performs the earlier described Step 2 to generate an attest for itself.

If, however, $\mathcal{D}_k$ has already attested itself in the current attestation phase ($curpid_i = curpid_k$), then both devices will exchange their attestation reports. For efficiency reasons, $\mathcal{D}_k$ will only initiate this exchange, if $\mathcal{D}_i$ announces an attestation proof for at least one device that is not already contained in $\mathcal{D}_k$'s report. $\mathcal{D}_k$ checks this by comparing the list of device ids $rep_i.ids$, already communicated in message msg_{ann}, with its own list of attested devices $rep_k.ids$. If a difference is identified, $\mathcal{D}_k$ requests for $\mathcal{D}_i$'s report by sending a message msg_{repreq} that also includes the device identifiers of all its stored attestation proofs $rep_k.ids$. This allows $\mathcal{D}_i$ to generate a subset *srep* of its own stored report (with the for now unspecified function $\mathsf{SliceRep}()$) that only contains the attestation proofs that $\mathcal{D}_k$'s report is missing. After generating *srep*, $\mathcal{D}_i$ authenticates *srep* with a MAC mac_{ik}, which is computed by the function $\mathsf{GenMac}()$. The key mk_{ik} to compute this MAC is derived from the channel key ck_{ik} as well as the phase id $curpid_i$ using a Key Derivation Function (KDF). This form of authentication prevents Adv_{sw} from reusing MACs that were captured from other devices or previous attestation phases. Both *srep* and mac_{ik} are transmitted from $\mathcal{D}_i$ to $\mathcal{D}_k$ in a message msg_{rep}.

On receiving msg_{rep}, $\mathcal{D}_k$ recomputes the MAC key mac_{ik} and then verifies the received report *srep* with the function $\mathsf{VerMac}()$. This ensures that (i) *srep* has

not been manipulated during transmission, (ii) *srep* originates from $\mathcal{D}_i$, and (iii) $\mathcal{D}_i$ is healthy according to *vss* (as otherwise, $\mathcal{D}_i$ would not have set $curpid_i$ to *pid* in the previous step). If the verification is successful, $\mathcal{D}_k$ will aggregate *srep* with its local report rep_k. This functionality is realized by AggRep(), which stores the union of $rep_i.ids$ and $rep_k.ids$ as well as $rep_i.proofs$ and $rep_k.proofs$ in rep_k. Since this operation increases $\mathcal{D}_k$'s accumulated attestation proofs, $\mathcal{D}_k$ distributes an updated msg_{ann} to inform its neighboring devices.

(4) Attestation Finalization. Eventually, $\mathcal{O}$ connects to an arbitrary device $\mathcal{D}_i$ and request, receives, and verifies the attestation report of $\mathcal{D}_i$ using the same protocol as described in Step 3. The time $\mathcal{O}$ waits between Step 1 and Step 4 is dependent on the network size, network density, device mobility, security parameters, and device-to-device communication capabilities. $\mathcal{O}$ may conduct real-world experiments or simulations to estimate a proper waiting time. In a network with 3000 devices, a communication range of 50m, a deployment area of 5000m x 5000m, high network dynamics, and a high security level, we identified an optimal waiting time of approximately 20 minutes (Section 5.4.2). Next, $\mathcal{O}$ verifies all attestation proofs contained in the received report using VerRep(), as described in Section 5.3. Thereby, attacks where Adv_{hw} attempts to fake a healthy system state for devices with a compromised software, but healthy hardware, are detected. If both checks pass, $\mathcal{O}$ considers all devices listed in the report to be healthy. Depending on $\mathcal{O}$'s goals, that is, verifying specific devices, the majority of devices, or all devices, $\mathcal{O}$ either ends the attestation phase or waits for more devices to contribute to the aggregated report.

To end the attestation phase, $\mathcal{O}$ sends a signed finish message containing the phase id *pid* to $\mathcal{D}_i$. Devices that receive the finish message verify its authenticity, stop executing Step 3, and propagate the finish message to neighboring devices that still execute Step 3. We note that the attestation result may exhibit false positives, as $\mathcal{O}$ cannot be sure whether all healthy devices indeed contributed to the received report. However, it is impossible to distinguish between a compromised and an absent device, i.a., as compromised device can intentionally not respond during attestation. Nevertheless, $\mathcal{O}$ can increase the probability that all healthy devices are contained in the attestation result by not emitting a finish message and repeating Step 4 at a later time, thereby prolonging the waiting time.

Remarks. For clarity, we omitted implementation details. In practice, timeouts are required to prevent the protocol from halting upon manipulations or errors. Furthermore, it is more efficient to only transmit the quantity and hash value of device identifiers in the announcement messages (msg_{ann}), instead of actually transmitting all identifiers from the report. Moreover, when receiving an announcement message msg_{ann} from a $\mathcal{D}_i$, it is useful to not only check for proofs to receive but also to check for proofs that $\mathcal{D}_i$ is missing, in order to initiate an exchange of these. Additionally, the retransmission of the same msg_{ann} to a particular device should be avoided, e.g., when a device is encountered repeatedly.

Security. As specified in our adversary model (Section 2.4), we refer to an attestation protocol as secure, if an adversary is unable to report a valid attest, i.e., a healthy system state, for a device that is at the time of its own attestation in a compromised state. To pass the attestation phase with a compromised device $\mathcal{D}_A$, an adversary must (i) generate an attestation report rep_A that contains a valid attestation proof for $\mathcal{D}_A$, and (ii) transmit rep_A with a valid msg_{rep} to a healthy device or the network operator $\mathcal{O}$.

Adv_{sw}, who cannot break the security of the TEE, is unable to achieve both (i) and (ii). Since all protocol code is executed inside the TEE, verifying the software integrity on compromised devices fails in Step 2, so that compromised devices refuse to update their phase id $curpid$ and to generate a report (failing (i)). Due to the outdated $curpid$, compromised devices construct an invalid MAC key mk in Step 3, hence, are unable to generate a msg_{rep} that contains a valid MAC and is accepted by healthy devices (failing (ii)).

By contrast, Adv_{hw} can achieve (ii), since Adv_{hw} can break into the TEE of a device, gain access to a valid MAC key mk for the current attestation phase, and thus synthesize valid msg_{rep} messages that are accepted by other healthy devices. To achieve (i) and generate a valid attestation proof for $\mathcal{D}_A$, Adv_{hw} requires access to the attestation key ak_A of $\mathcal{D}_A$. However, Adv_{hw} has only access to the attestation key of devices that are physically tampered with. Thus, Adv_{hw} can only forge valid attestation proofs for physically compromised devices, but no other devices. Since physical attacks require a lot of resources and do not scale (Section 2.4), SALAD effectively mitigates physical attacks.

Moreover, Denial of Service (DoS) attacks in the network only affect devices whose communication is directly disturbed, e.g., jammed by Adv_{sw}. All other devices can successfully participate in the attestation.

5.3 ATTESTATION REPORT AGGREGATION

In the following, we first describe a contemporary basic solution to represent attestation reports and apply it to SALAD (Section 5.3.1). Afterwards, we present two aggregation schemes, which aim at minimizing the message and storage overhead entailed by the basic scheme (Section 5.3.2). Finally, we propose an approach that allows the parameterization of security and performance in the proposed schemes (Section 5.3.3).

5.3.1 *Basic MAC Scheme*

MACSimple. In the straight forward solution, attestation reports store the attestation proofs of individual devices unaggregated, as a list of signatures or Message Authentication Codes (MACs) [28]. On embedded devices, MACs are preferred over signatures for efficiency reasons. We adapt this approach to SALAD in a scheme that is henceforth referred to as MACSimple. Recall that in SALAD an

attestation report *rep* consist of two parts: (1) *rep.ids*, a list of device identifiers describing all devices that contributed to the attestation report; (2) *rep.proofs*, a data structure that stores the attestation proofs of all devices listed in *rep.ids*. In MACSimple, *rep.ids* and *rep.proofs* are both lists. In the following, we describe all operations performed by MACSimple.

GenRep(ak_i; $curpid_i$; sp): To generate a new report, $\mathcal{D}_i$ sets *rep.ids* to a list that only contains its identifier i. Furthermore, $\mathcal{D}_i$ computes its attestation proof by generating a MAC over $curpid_i$ with its attestation key ak_i (GenMac(ak_i; $curpid_i$)). Afterwards, *rep.proofs* is set to a list that only contains the computed MAC. The security parameter sp defines the settings of the MAC algorithm.

AggRep($repA$; $repB$): Two reports are aggregated by concatenating both lists, i.e., the device ids ($repA.ids$ and $repB.ids$) and the attestation proofs ($repA.proofs$ and $repB.proofs$). Afterwards, duplicates within each list are removed.

SliceRep($repA$; $repB.ids$): To generate a subset $srep$ of $repA$ that specifically contains no attestation proofs from devices listed in $repB.ids$, $srep.ids$ is set to the relative complement of $repB.ids$ in $repA.ids$ ($srep.ids = repA.ids \setminus repB.ids$). In addition, the appropriate attestation proofs from $repA.proofs$ are copied to $srep.proofs$, taking into account their order in $srep.ids$.

VerRep(pid; $\mathsf{ak}_i \ldots \mathsf{ak}_n$; rep): $\mathcal{O}$ verifies a report by recomputing the MAC for each device contained in *rep.ids* and checking whether the recomputed MACs match the particular MACs in *rep.proofs*. If this is the case, $\mathcal{O}$ assumes that all devices listed in *rep.ids* are in a healthy system state.

To minimize the size of *rep.ids*, device ids are represented in three different ways, depending on the number of entries: as a list of ids present in the report, as a list of device ids not present in the report, or as an n-bit vector where a one at position i indicates that $\mathcal{D}_i$ is in the report.

5.3.2 *Extended MAC Aggregation Schemes*

Motivation. With the basic scheme (Section 5.3.1) attestation reports are relatively large, as they contain an individual MAC from each device that contributed to the report. To reduce the size of *rep.proofs*, recent protocols employ tree-based aggregation schemes that compress multiple signatures or MACs into a small aggregate [7, 68]. In these protocols, devices form a static tree topology, in which each node aggregates its own report with the reports of its children, before transmitting the aggregated report to its parent. Since this approach is uneconomical in dynamic or even infeasible in disruptive networks, SALAD pursues a distributed approach, where devices pass their own attestation proof to multiple devices. Hence, devices may need to aggregate reports that contain an *intersection*, i.e., at least one proof from the same device is present in both reports.

Figure 5.2 illustrates the occurrence of an intersection for a small exemplary network. The figure shows the attestation of six devices over five time periods. In each time period, all devices that meet exchange their aggregates. Starting

Time	*Comm.*	*Scheme*	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	$\mathcal{D}_5$	$\mathcal{D}_6$
$t=1$	$\mathcal{O} \leftrightarrow \mathcal{D}_1$	MACGreedy	$\boxed{\mathcal{D}_1}$					
		MACSmart	$\boxed{\mathcal{D}_1}$					
$t=2$	$\mathcal{D}_1 \leftrightarrow \mathcal{D}_2$	MACGreedy	$\boxed{\mathcal{D}_1, \mathcal{D}_2}$	$\boxed{\mathcal{D}_1, \mathcal{D}_2}$				
		MACSmart	$\boxed{\mathcal{D}_1}\boxed{\mathcal{D}_2}$	$\boxed{\mathcal{D}_2}\boxed{\mathcal{D}_1}$				
$t=3$	$\mathcal{D}_1 \leftrightarrow \mathcal{D}_3$ $\mathcal{D}_2 \leftrightarrow \mathcal{D}_4$	MACGreedy	$\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3}$	$\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_4}$	$\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3}$	$\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_4}$		
		MACSmart	$\boxed{\mathcal{D}_1}\boxed{\mathcal{D}_2}\boxed{\mathcal{D}_3}$	$\boxed{\mathcal{D}_2}\boxed{\mathcal{D}_1}\boxed{\mathcal{D}_4}$	$\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3}$	$\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_4}$		
$t=4$	$\mathcal{D}_1 \leftrightarrow \mathcal{D}_2$ $\mathcal{D}_3 \leftrightarrow \mathcal{D}_6$ $\mathcal{D}_4 \leftrightarrow \mathcal{D}_5$	MACGreedy	$\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3}$ $\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_4}$	$\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_4}$ $\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3}$	$\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_6}$	$\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_4, \mathcal{D}_5}$	$\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_6}$	
		MACSmart	$\boxed{\mathcal{D}_1}\boxed{\mathcal{D}_2}\boxed{\mathcal{D}_3}$ $\boxed{\mathcal{D}_4}$	$\boxed{\mathcal{D}_2}\boxed{\mathcal{D}_1}\boxed{\mathcal{D}_4}$ $\boxed{\mathcal{D}_3}$	$\boxed{\mathcal{D}_3}\boxed{\mathcal{D}_1}\boxed{\mathcal{D}_2}\boxed{\mathcal{D}_6}$	$\boxed{\mathcal{D}_4}\boxed{\mathcal{D}_1}\boxed{\mathcal{D}_2}\boxed{\mathcal{D}_5}$	$\boxed{\mathcal{D}_5}\boxed{\mathcal{D}_1}\boxed{\mathcal{D}_2}\boxed{\mathcal{D}_4}$	$\boxed{\mathcal{D}_6}\boxed{\mathcal{D}_1}\boxed{\mathcal{D}_2}\boxed{\mathcal{D}_3}$
$t=5$	$\mathcal{D}_5 \leftrightarrow \mathcal{D}_6$	MACGreedy	$\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3}$ $\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_4}$	$\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_4}$ $\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3}$	$\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_6}$	$\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_4, \mathcal{D}_5}$	$\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_6}$ $\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_4, \mathcal{D}_5}$	$\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_6}$ $\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_4, \mathcal{D}_5}$
		MACSmart	$\boxed{\mathcal{D}_1}\boxed{\mathcal{D}_2}\boxed{\mathcal{D}_3}$ $\boxed{\mathcal{D}_4}$	$\boxed{\mathcal{D}_2}\boxed{\mathcal{D}_1}\boxed{\mathcal{D}_4}$ $\boxed{\mathcal{D}_3}$	$\boxed{\mathcal{D}_3}\boxed{\mathcal{D}_1}\boxed{\mathcal{D}_2}\boxed{\mathcal{D}_6}$	$\boxed{\mathcal{D}_4}\boxed{\mathcal{D}_1}\boxed{\mathcal{D}_2}\boxed{\mathcal{D}_5}$	$\boxed{\mathcal{D}_5}\boxed{\mathcal{D}_1}\boxed{\mathcal{D}_2}\boxed{\mathcal{D}_4}$ $\boxed{\mathcal{D}_1}\boxed{\mathcal{D}_2}\boxed{\mathcal{D}_3}\boxed{\mathcal{D}_6}$	$\boxed{\mathcal{D}_6}\boxed{\mathcal{D}_1}\boxed{\mathcal{D}_2}\boxed{\mathcal{D}_3}$ $\boxed{\mathcal{D}_1}\boxed{\mathcal{D}_2}\boxed{\mathcal{D}_4}\boxed{\mathcal{D}_5}$

Figure 5.2: Attestation report aggregation with the MACGreedy and MACSmart aggregation schemes. Column *Time* shows an abstract time period, *Comm.* illustrates devices that meet and exchange reports, *Scheme* annotates the aggregation scheme. Columns $\mathcal{D}_1$ to $\mathcal{D}_6$ show the aggregated attestation proofs stored by the particular device. Intersecting aggregates are highlighted in red.

with time $t = 1$, $\mathcal{O}$ invokes the attestation protocol by notifying $\mathcal{D}_1$, which in $t = 2$ exchanges aggregates with $\mathcal{D}_2$. In $t = 3$, $\mathcal{D}_1$ and $\mathcal{D}_3$ meet, as well as $\mathcal{D}_2$ and $\mathcal{D}_4$. At $t = 4$, devices $\mathcal{D}_1$ and $\mathcal{D}_2$ meet for the second time. Yet, now they cannot merge their aggregated proofs, since $\mathcal{D}_1$ has a proof for $\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3}$ and $\mathcal{D}_2$ has a proof for $\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_4}$ that both contain an attestation proof for $\mathcal{D}_1$ and $\mathcal{D}_2$. In simulations, we discovered that these intersections frequently occur in dynamic and disruptive networks (Section 5.4). MACSimple can union intersecting reports because it stores each proof individually, i.e., unaggregated. By contrast, advanced aggregation schemes are unable to resolve intersections, as their aggregates cannot be reverted back into individual proofs. For instance, protocols that use MAC-based attestation proofs [68] perform an XOR-operation for aggregation [77]. Hence, intersections cancel each other out upon aggregation, e.g., $\mathcal{D}_1$ and $\mathcal{D}_2$ would store the (incomplete) aggregate $\boxed{\mathcal{D}_3, \mathcal{D}_4}$ after performing a XOR-operation at $t = 4$.

To take advantage from MAC aggregation [77] in SALAD, we propose two extended MAC aggregation schemes that can handle intersecting attestation reports. Both schemes require attestation reports to store not only one aggregated attestation proof and device identifier list, as in MACSimple, but *multiple* tuples of attestation proofs and device identifier lists. Consequently, each report stores l report tuples, where a tuple with index $1 \leq i \leq l$ consists of a *single* aggregated MAC $rep.proofs_i$ and a device identifier list $rep.ids_i$. Each $rep.ids_i$ indicates the identity of all devices whose attestation proof is aggregated in $rep.proofs_i$.

MACGreedy. Our first MAC-based aggregation scheme, named MACGreedy, aims at minimizing the storage consumption of attestation reports by aggregating all MACs in a greedy manner, using a XOR aggregation. More specifically, MACGreedy attempts to immediately aggregate any report tuple of newly received reports with all stored report tuples. If a received report tuple can, due to intersections, not be aggregated with any stored tuple, it is appended as a new tuple to the stored report. As a result, all report tuples stored in *rep* are intersecting and cannot be further aggregated. Moreover, MACGreedy removes all tuples that are subsets of other tuples, to further reduce the size of the overall report. Algorithm 5.1 depicts the precise pseudocode for aggregating two reports. Generating and verifying attestation reports is similar to MACSimple, despite each tuple in the report must be processed individually. To generate a subset report of *rep* that specifically covers certain devices with SliceRep(), MACGreedy finds a minimum combination of tuples in *rep* that cover the requested device identifiers. Unfortunately, finding this combination is an NP-complete problem, namely, a variation of the minimum set cover problem. In our implementation (Section 5.4), we solve this problem by initially selecting all tuples that are the only tuples covering a request device identifier. Afterwards, a greedy heuristic iteratively selects the tuple that covers the most requested device identifiers, until all device identifiers are covered by the selected tuples. This approach turned out to provide sufficient solutions in comparatively short time (Section 5.4).

Algorithm 5.1: Pseudocode of AggRep() function in MACGreedy.

```
1:  procedure AggRep(repA; repB)
2:      for i = 1, 2, ..., len(repB) do
3:          merged ← false
4:          for k = 1, 2, ..., len(repA) do
5:              if (repB.ids_i ∩ repA.ids_k) = ∅ then
6:                  repA.proofs_k ← repA.proofs_k ⊕ repB.proofs_i
7:                  repA.ids_k ← repA.ids_k ∪ repB.ids_i
8:                  merged ← true
9:                  break
10:         if merged = false then
11:             repA.proofs_{1+len(repA)} ← repB.proofs_i
12:             repA.ids_{1+len(repA)} ← repB.ids_i
13:     for i = 1, 2, ..., len(repA) do
14:         for k = 1, 2, ..., len(repB) do
15:             if i ≠ k and repA.ids_i ⊂ repA.ids_k then
16:                 repA.proofs ← repA.proofs \ repA.proofs_i
17:                 repA.ids ← repA.ids \ repA.ids_i
18:     return repA
```

MACSmart. The objective of our second MAC aggregation scheme, MACSmart, is to minimize the size of transmitted attestation reports. For this purpose, MACSmart stores received report tuples separately, and only aggregates (XORs) them for transmission, which reduces communication costs. We remark that this approach is noticeably different to storing individual proofs (as in MACSimple), since transmitted tuples, i.e., also received tuples, are aggregated by the sending device to the best of its abilities. In detail, AggRep(*repA*,*repB*) generates a new report that contains all tuples of *repA* and *repB*. Only when an attestation report is prepared for transmission with SliceRep(), MACSmart aggregates stored tuples in an optimal way to cover specific attestation proofs and their device identifiers. Like in MACGreedy, this requires MACSmart to solve a variation of the set cover problem. However, in addition, MACSmart must also solve a variation of the set packing problem, to find an optimal combination of tuples that are non-intersecting, hence, can be aggregated (in MACGreedy this is redundant as all stored tuples are intersecting). We implemented a heuristic that is able to solve this issue in reasonable time on low-end embedded devices (Section 5.4). The heuristic initially constructs a graph in which tuples are represented as vertices and all non-intersecting tuples are connected by edges. Then, we employ the Bron-Kerbosch algorithm [24] to find maximal cliques in the graph. A clique is a subset of vertices where each vertex is connected to all other vertices by an edge. Accordingly, each clique in our graph is a set of non-intersecting tuples, hence, can be aggregated to a single tuple. Our heuristic constructs the final result by

iteratively determining the largest (remaining) clique, aggregating its tuples, and appending the aggregate to the result. Using this approach, MACSmart achieves a stronger aggregation with fewer intersections than MACGreedy, albeit at a higher storage overhead. Figure 5.2 demonstrates that MACSmart enables $\mathcal{D}_1$ and $\mathcal{D}_2$ to exchange $\boxed{\mathcal{D}_4}$, respectively $\boxed{\mathcal{D}_3}$, at $t = 4$. When requested for their full report in $t = 4$, this would allow $\mathcal{D}_1$ and $\mathcal{D}_2$ to assemble the optimal report $\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_4}$, which contains only a single aggregated proof. At the same time, MACGreedy requires $\mathcal{D}_1$ and $\mathcal{D}_2$ to assemble a report that contains two attestation proofs, namely, $\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3}$ and $\boxed{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_4}$. Nevertheless, as shown in $t = 5$, MACSmart cannot prevent the arising of intersecting tuples. Hence, with many devices, transmitted reports still contain evermore tuples that cannot be aggregated.

5.3.3 *Trading Security for Performance*

MAC Truncation. The extended MAC aggregation schemes presented in the last section (Section 5.3.2) are able to decrease the communication costs of MACSimple by up to 40%. Nevertheless, in networks with high dynamics or many devices, their entailed communication, runtime, or storage costs may still be too high (Section 5.4). For these cases, we propose that, instead of using the standard MAC size [68], aggregation schemes truncate MACs to save overhead. Thereto, we parameterize the aggregation schemes $\mathsf{MACSimple}(t)$, $\mathsf{MACGreedy}(t)$, $\mathsf{MACSmart}(t)$, which operate like before, but shorten the attestation proofs, i.e., MACs, to t bits. The parameter t is chosen by the network operator $\mathcal{O}$ in each attestation phase. Because certain MAC algorithms are vulnerable to truncation attacks [147], we suggest to use Hash-based Message Authentication Codes (HMAC) [92]. Accordingly, a device $\mathcal{D}_i$ generates its attestation proof by using the t leftmost bits of an HMAC computed over $curpid_i$ with the key ak_i ($\mathsf{HMAC}(\mathsf{ak}_i, curpid_i)$). For security, the HMAC specification recommends to set t to at least 80 bits [92]. Yet, in the following, we argue that when using $\mathsf{MACSimple}(t)$, a $t < 80$ bits can be a reasonable choice in many use cases of attestation.

Security. Due to the authenticated exchange of attestation reports, Adv_{sw} is unable to contribute an attest to any report. Hence, changing t only affects the security against Adv_{hw}, but not Adv_{sw}.

Moreover, devices use their secret attestation key to compute the HMAC, which is unknown to an adversary. Thus, collision attacks on the HMAC are irrelevant and only preimage attacks need to be considered. Using an output size of n bits typically provides n bits security against preimage attacks, as opposed to $n/2$ bits against collision attacks (due to birthday attacks). Hence, to provide a high security level of 128 bits, it is sufficient to set $t = 128$.

Furthermore, t itself has no influence on the preimage resistance of the HMAC. Using a low t increases the probability with which Adv_{hw} can successfully forge the HMAC, but not the probability with which Adv_{hw} can break the preimage

t	Forged MACs	$\Pr[Adv_{hw} wins]$	Devices	Report Size
1	1	0.5	100	25 bytes
1	10	0.000977	1000	250 bytes
10	1	0.000977	100	138 bytes
10	10	$7.89 \cdot 10^{-31}$	1000	1375 bytes
20	1	$9.54 \cdot 10^{-7}$	100	263 bytes
20	10	$6.22 \cdot 10^{-61}$	1000	2625 bytes
80	1	$8.27 \cdot 10^{-25}$	100	1013 bytes
128	1	$2.94 \cdot 10^{-39}$	1000	16125 bytes

Table 5.2: Statistical security and size of attestation report for MACSimple(t) with different truncate parameter t (bits), number of forged MACs from Adv_{hw}, and number of devices in the network.

resistance, thus, reconstruct attestation keys. In short, Adv_{hw} has no advantage over guessing an HMAC, which is successful with the probability $1/2^t$.

In MACSimple(t), this probability of successfully forging an attestation proof only holds for one device in a single run of the attestation phase. Recall that Adv_{hw} requires significant resources to physically compromise a few devices. Therefore, in practice, it may be enough to prevent Adv_{hw} from faking a healthy system state for many devices, and/or for the same device over multiple attestation rounds, in order to render physical attacks uneconomical and deter Adv_{hw}. This is especially the case when $\mathcal{O}$ only needs to determine whether the majority of devices are in a healthy state. However, the probability of success for Adv_{hw} exponentially decreases with the number of forged attestation proofs and rounds, rapidly becoming negligible. Thus, the goal of preventing any form of scalable attacks is achieved even for very low t, e.g., $t = 20$. Table 5.2 provides precise numbers for different configurations. Note that in MACGreedy and MACSmart, truncated MACs have more severe security implications. This is due to their MAC aggregation, which allows Adv_{hw} to successfully fake an attestation proof for multiple devices by forging only a single MAC.

Furthermore, we would like to emphasize that the attestation phase can only be initiated and verified by $\mathcal{O}$. Adv_{hw} has no indications whether a forged attestation report is valid or invalid, before handing it to $\mathcal{O}$. However, if the verification of the attestation report fails, $\mathcal{O}$ can take actions like increasing t or physically inspecting devices in the network to look for manipulations. For this reason, forging an attestation proof may not be required to be as hard as breaking common cryptographic primitives.

In practice, t needs to be adjusted to the particular use case. Yet, due to the provided reasons as well as the security and performance insights from Table 5.2 and our evaluation (Section 5.4), we observe that MACSimple(20) provides an

Scheme	Operation	Runtime
ed25519	GenKey()	18.26 ms
	KeyExchange()	48.84 ms
	VerSig(32 B)	50.67 ms
SHA-256	Hash(32 B)	0.08 ms
	Hash(32 kB)	40.02 ms
SHA-256 HMAC	GenMAC/VerMAC(32 B)	0.23 ms
	GenMAC/VerMAC(32 kB)	39.86 ms

Table 5.3: Runtime of cryptographic algorithms on the Stellaris board.

adequate security for typical use cases while increasing performance by more than an order of magnitude.

5.4 EVALUATION

In this section, we first describe our implementation and measurements (Section 5.4.1). Then, we present our simulation setup and the results of our network simulations (Section 5.4.2).

5.4.1 *Implementation and Measurements*

Implementation. As a target platform for our implementation, we employed three Stellaris LM4F120H5QR microcontrollers from Texas Instruments that were equipped with CC2530 BosterPacks from Anaren for ZigBee wireless networking capabilities in the 2.4 GHz band. The Stellaris microcontrollers feature an 80 MHz ARM Cortex-M4F core, 256 kB flash, and 32 kB SRAM memory. Furthermore, they provide the minimal hardware properties required for remote attestation [127]. Cryptographic primitives were implemented based on TinyCrypt [69] and Supercop [17]. In detail, we used SHA-256 as a cryptographic hash function, a Keyed-Hash Message Authentication Code (HMAC) based on SHA-256 as a MAC and KDF, and Elliptic Curve Diffie-Hellman (ECDH) as well as Edwards-Curve Digital Signature Algorithm (EdDSA) based on Curve25519 [18] for key exchanges and digital signatures.

For the implementation of our secure code update scheme (Section 4.2) and SCAPI (Chapter 6), we also employed the Stellaris as a target platform. Nevertheless, compared to the implementation of our secure code update scheme and SCAPI, we switched from SHA-512 with Supercop [17] to SHA-256 with TinyCrypt [69], which increased the performance of hash computations, as shown in the following.

100 Devices	MACSimple	MACGreedy	MACSmart
AggRep()	0.18 ms	1.91 ms	0.64 ms
SliceRep()	0.09 ms	2.59 ms	2.71 ms
500 Devices	**MACSimple**	**MACGreedy**	**MACSmart**
AggRep()	0.79 ms	29.89 ms	5.72 ms
SliceRep()	0.42 ms	24.01 ms	20.07 ms
1000 Devices	**MACSimple**	**MACGreedy**	**MACSmart**
AggRep()	2.55 ms	76.56 ms	13.83 ms
SliceRep()	1.25 ms	52.05 ms	45.75 ms
1500 Devices	**MACSimple**	**MACGreedy**	**MACSmart**
AggRep()	4.42 ms	137.35 ms	20.44 ms
SliceRep()	2.17 ms	94.40 ms	73.33 ms

Table 5.4: Runtime of aggregation schemes on the Stellaris board.

Runtime Measurements. Table 5.3 shows runtime measurements of the implemented cryptographic algorithms used in SALAD. All operations that devices frequently perform during attestation, that is, authenticating and verifying messages, consume very little runtime (< 1 ms). Only signature verification (to verify $\mathcal{O}$'s attestation request), attesting the device's software (hash over complete firmware), and performing a key exchange (introducing a new device to the network), consume more runtime, with around 50 ms per operation, which is still practical.

The generation of reports (GenRep()) comes at negligible cost, but their aggregation (AggRep()) and partitioning (SliceRep()) can be computationally expensive, depending on the used aggregation scheme. We experimentally evaluated the runtimes of all proposed aggregation schemes in networks of different sizes. Table 5.4 illustrates averaged runtime measurements of the procedures AggRep() and SliceRep() for 100, 500, 1000, and 1500 devices. The table shows that MACSimple provides the best runtime. This is because only comparatively inexpensive operations are required (cf. binary search is its most expensive operation). By contrast, MACGreedy and MACSmart perform complex algorithmic tasks (Section 5.3). This involves solving an NP-complete problem for which we implemented a fast greedy heuristic. AggRep() is faster in MACSmart than in MACGreedy because MACGreedy attempts to aggregate all report tuples immediately, whereas MACSmart only aggregates report tuples in SliceRep().

Network Measurements. In Section 4.2.3 we already showed network measurements for Stellaris LM4F120H5QR microcontrollers that were equipped with CC2530 BosterPacks from Anaren for ZigBee wireless networking capabilities.

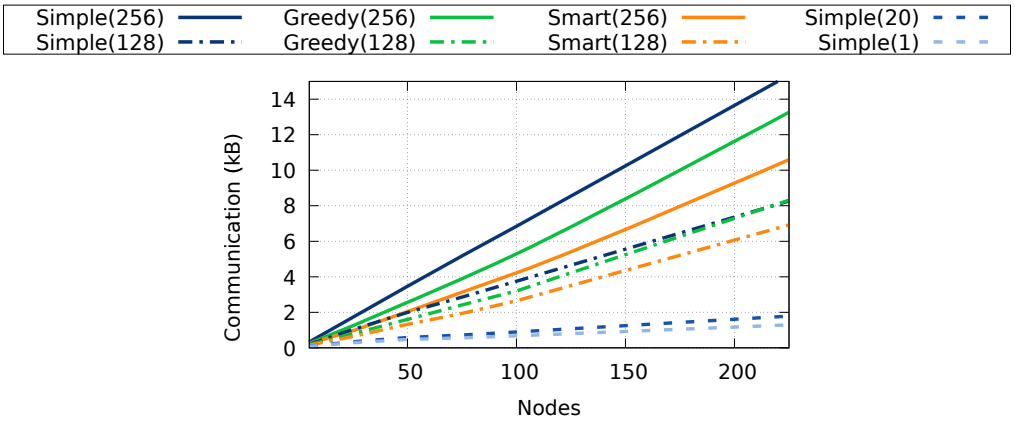
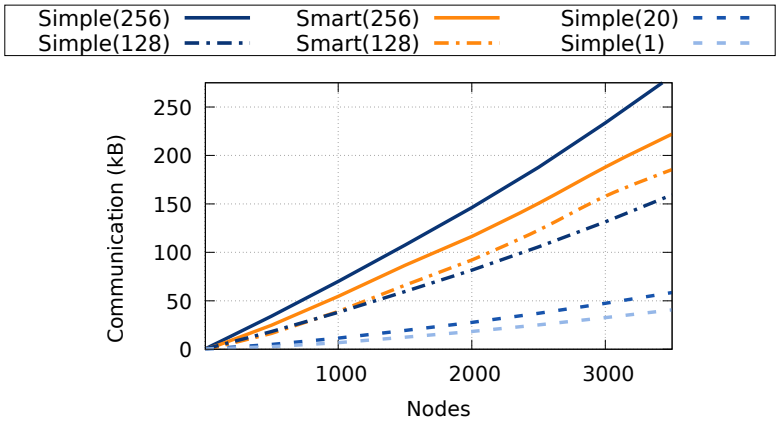


Figure 5.3: Average communication costs per device in networks of different sizes and with low (1-2 m/s) network dynamics.

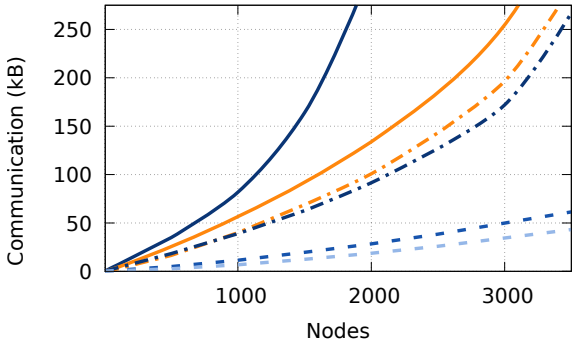
5.4.2 *Simulation Setup and Results*

Simulation Setup. To simulate SALAD in large networks, we used the OMNeT++ [146] event simulator. We implemented SALAD at the application layer and configured delays based on our runtime and network measurements. On lower network layers, we applied a simplified communication model that enables unimpaired communication whenever devices are within communication range. Two devices were only allowed to communicate half-duplex and while other devices within reach were inactive. In this respect, the granularity of our communication model lies between typical models used for the simulation of sparse (DTN) and dense (MANET) network topologies. This allows us to provide more realistic results than DTN simulators [80] while being able to scale simulations to a few thousands devices, which is infeasible with MANET simulators [21].

In general, we set the firmware size that is attested to 30 kB and the device communication range to 50m, being half of the distance specified in ZigBee. A simulation run starts with the network operator sending an attestation initiation to the first device and ends with all devices sharing the same attestation result. Each data point represents the average of 8 simulations with different seed values. At the start of each simulation, devices are randomly deployed in a 5.000m x 5.000m area and then move according to the random waypoint mobility model, which is one of the most popular mobility models to evaluate DTNs and MANETs. Consequently, devices repeatedly select a random destination within the area and then move towards this destination at a specified speed. In contrast to actual mobility traces that enable to examine the performance in specific application scenarios, synthetic mobility models, like the random waypoint mobility model, are well-suited to assess the general performance. We acknowledge that the random waypoint mobility model leads to a nonuniform spatial distribution of devices in the area [20], where devices are more likely to be located in the center of the area, as opposed to its edges. Nevertheless,



(a) Large networks with low dynamics (1-2 m/s).



(b) Large networks with high dynamics (10-20 m/s).

Figure 5.4: Average communication costs per device in networks of different sizes and with low (1-2 m/s) or high (10-20 m/s) network dynamics.

SALAD is free from assumptions on the spatial uniformity of devices. Thus, for the general conclusions that can be derived from simulations it is irrelevant whether the random waypoint or a more uniform mobility model is used.

Communication Costs. Figure 5.3 and 5.4 depict the average application layer traffic per device (sent and received) in medium-sized networks with up to 250 devices (Figure 5.3) and larger networks with up to 3500 devices (Figure 5.4). The simulation shows that communication costs are strongly dependent on the used aggregation scheme and its parameters. This is because communication during attestation is dominated by the exchange of attestation reports (msg_{rep}), which are represented according to the selected aggregation scheme (Section 5.3).

In medium-sized networks (Figure 5.3), MACGreedy and MACSmart outperform MACSimple by requiring considerably less communication. In detail, when SHA256-HMACs are not truncated, MACSmart reduces the communication costs of MACSimple by 42.2% with 50 devices or 31.8% with 200 devices. In general, MACGreedy performs worse than MACSmart, as its aggregation strategy is optimized for storage rather than communication.

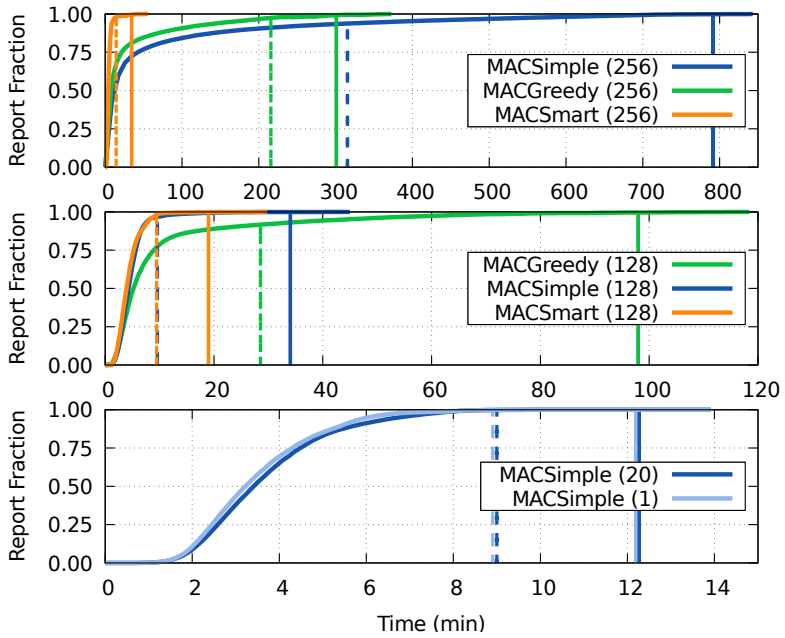


Figure 5.5: Fraction of final report that an average device stores at a given time in a network with 3000 devices and high dynamics (device speed 10-20 m/s). The average time till the first/last network device stores the final report is indicated by vertical dotted/solid lines.

A very efficient way to further reduce communication cost is to truncate the SHA256-HMAC, which is indicated for all schemes by parameter $t < 256$. Recall that the chances of Adv_{hw} to successfully forge an attestation report increases with a decreasing t. Nevertheless, in Section 5.3.3, we showed that MACSimple(20) provides sufficient security for typical attestation use cases (but not MACGreedy(20) or MACSmart(20)). We observe that communication costs can be reduced by almost a factor of two for MACSmart(128) and MACGreedy(128) and by up to one order of magnitude for MACSimple(20). Since MACSimple communicates MACs in an unaggregated form, it benefits more from truncated MACs than MACSmart and MACGreedy, as shown by $t = 128$ in Figure 5.3 and 5.4.

Investigating the more promising schemes MACSimple and MACSmart in larger networks, we see that MACSmart outperforms MACSimple for the standard MAC size, but for all truncation parameters $t \in \{1, 20, 128\}$, MACSimple entails less communication. We also observe that significantly more communication is required with high (Figure 5.4b) than with low network dynamics (Figure 5.4a). Reason for this are connection breaks, which occur more frequently with high dynamics. Whenever devices communicate but are unable to exchange an attestation report successfully due to connection disruptions, communication is wasted. The probability that a report exchange fails increases when devices move fast (high dynamics) and attestation reports are big (large networks). This is why aggregation techniques that lead to smaller reports suffer less from high network dynamics. Consequently, as shown in Figure 5.4, MACSimple(1) and

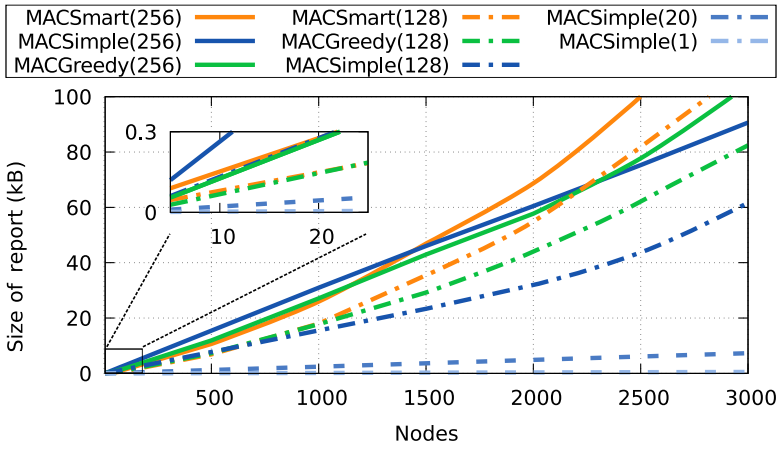


Figure 5.6: Average per device storage consumption of attestation reports.

MACSimple(20) perform only slightly worse in high, compared with low, network dynamics. We also varied the network density by enlarging or reducing the deployment area but found no impact on the communication costs.

Attestation Runtime. In smaller networks, we identified that the selected aggregation scheme has only little influence on the attestation runtime and communication or computational optimizations will not be able to further reduce the runtime. This is because SALAD already pursues the fastest approach of achieving a common attestation result on all devices, as attestation proofs are propagated in an epidemic manner. Instead, we found the major bottleneck for the attestation runtime in the lack of communication opportunities between devices. Thus, in smaller networks, attestation runtimes are heavily dependent on the underlying mobility model.

Nevertheless, in highly dynamic networks with many devices, attestation reports become large and devices may have too little contact time to successfully exchange their reports. In these networks, attestation runtime can be significantly improved by using an aggregation scheme that leads to more compact reports, thereby, increases the probability of success for report exchanges. Figure 5.5 illustrates this effect in a network with 3000 devices and high network dynamics. The figure depicts the average number of devices, measured as a fraction of all devices, that contributed to the report at a particular time. As shown, compared with the basic aggregation scheme MACSimple, attestation runtime can be decreased by more than an order of magnitude using MACSmart, or almost two orders of magnitude when truncating MACs with MACSimple(20). Thus, in large dynamic networks, only MACSmart(128) and MACSimple($t \leq 20$) achieve a complete attestation with a practical runtime in the range of 10 to 20 minutes. If it is sufficient to merely verify a fraction of all devices, the attestation result can be obtained much earlier. For instance, the majority of devices can be verified in less than one third of the time required to verify all devices.

Storage. The vast majority of storage is consumed by channel keys (ck) and attestation reports. Whereas channel keys consume $16n$ bytes on each device, with n being the number of network devices, the storage consumption of attestation reports depends on the selected aggregation scheme. Figure 5.6 shows the storage consumption of the final attestation report on an average device in the network. It illustrates that our extended schemes, MACGreedy and MACSmart consume less storage than MACSimple in networks up to roughly 600 devices when truncating MACs to 128 bits, or in all networks with untruncated MACs. Although MACGreedy was designed with storage consumption in mind, it unfortunately performs in most cases only a little better than MACSmart. Again, by truncating MACs, the overhead can be radically reduced, for instance, by more than two orders of magnitude when comparing MACSimple(256) with MACSimple(1). Note that transmitted reports are much smaller than stored reports, as SliceRep() ensures that only fractions of the actual report are transmitted during attestation.

Summary. Our evaluation revealed that the selected aggregation scheme has a big impact on the attestation phase. Using MACSmart, the communication and runtime overhead can be reduced by up to 40% and 95%, compared with the basic scheme MACSimple. However, it is even more efficient to apply MACSimple(t) and truncate MACs to an output length of $t < 128$ bits. For instance, MACSimple(20), provides a security level that is suitable for typical attestation use cases while reducing communication, runtime, and storage costs by more than an order of magnitude. This enables SALAD to be a secure and lightweight attestation scheme for highly dynamic and disruptive networks.

5.5 SUMMARY

In this chapter, we presented SALAD, a collective attestation protocol for highly dynamic and disruptive networks. SALAD pursues a novel distributed approach, in which all devices exchange and accumulate proofs that attest the trustworthiness of devices in the network. This way, SALAD performs well in highly dynamic and disruptive networks, enables the verifier to obtain the attestation result from any network device, increases resilience to denial of service attacks, and also mitigates physical attacks. To reduce communication costs and attestation runtime, we presented aggregation schemes that compress the size of exchanged attestation proofs. Compared with a basic solution, our aggregation schemes are able to reduce the communication and runtime overhead by 40%, resp. 90%, for a very high security level, and by more than 90%, resp. 98%, for a security level sufficient for practical use cases. We demonstrated the security of SALAD and evaluated our protocol in network simulations based on measurements that we conducted on low-end embedded devices.

6 ATTESTATION OF SOFTWARE AND PHYSICAL ATTACKS

Traditional attestation protocols only protect against software attacks and consider physical attacks to be out of scope. Due to this reason, an adversary is able to corrupt their attestation results after physically tampering with the hardware of prover devices. Recently proposed protocols combine collective attestation with absence detection to detect software and physical attacks. However, these protocols suffer from limited scalability and robustness, which impairs their applicability in practice. In this chapter, we present a scalable attestation protocol that detects software and physical attacks. Based on the assumption that physical attacks require an adversary to capture and disable devices for a noticeable amount of time, our protocol identifies devices with compromised hardware and software. Compared to existing solutions, our protocol reduces communication complexity and runtimes by orders of magnitude, precisely identifies compromised devices, and is robust against failures or network disruptions. We demonstrate that our protocol is highly efficient in well-connected networks and operates robust in very dynamic network topologies.

Remark. Parts of this chapter have been published in [88].

6.1 MOTIVATION AND CONTRIBUTION

Embedded systems are frequently used in applications where they are publicly accessible, left unattended, and deployed in large quantities. These circumstances allow an adversary to physically approach and tamper with the hardware of embedded devices much easier than with the hardware of traditional computer systems. Collective attestation protocols (Section 2.1.2), which enable the efficient attestation of entire networks, commonly only protect against software attacks [7, 11, 28, 44, 65, 67]. Thus, by physically compromising devices, an adversary can corrupt their attestation results and fake a healthy system state for devices that are actually in a compromised state.

DARPA [68] represents the first approach to solve this problem by combining existing scalable attestation approaches [7, 11] with absence detection [32] to detect both software and physical attacks. Absence detection builds on the assumption that an adversary, who physically tampers with a device, must temporarily take the device offline for a noticeable amount of time, e.g., to disassemble the device and extract secret keys [14]. To detect offline and thus physically compromised devices, each device periodically emits a heartbeat that needs to be received, verified, and logged by every other device in the network. Although a functional solution to the problem, the protocol suffers from several shortcomings. First, the amount of exchanged messages per periodically

executed heartbeat period scales quadratically with the number of devices in the network. This causes scalability issues in large networks with respect to network communication, energy consumption, and runtime performance. Furthermore, the protocol is error-prone, since a single defective transmission of a heartbeat suffices to cause a false positive, where a healthy device is mistakenly regarded as compromised. Aggravating this, the protocol is only able to attest the state of the overall network and cannot identify particular compromised devices. Hence, a single false positive causes the entire network to be mistakenly regarded as compromised. Finally, the protocol relies on the assumption that the network topology is static and connected, which is a strong limitation.

Contribution. In this chapter, we present SCAPI, a scalable attestation protocol that detects software and physical attacks. To detect physical tampering, SCAPI builds on the established assumption that an adversary needs to take a device offline to physically tamper with it [32, 33, 65, 68]. During protocol execution, healthy prover devices maintain a secret communication key that compromised devices are unable to obtain, which excludes compromised devices from the network. This approach is similar to our secure code update scheme (Section 4.2), where the key is, however, only dependent on the software integrity of prover devices. By contrast, in SCAPI, the key is dependent on the hardware integrity, whereas the software integrity is determined using existing scalable software attestation approaches [7, 11].

In detail, all physically healthy devices share a common session key that is periodically regenerated by a leader device and propagated in the network. To transmit the newest session key from one device to the other, sender and receiver must mutually authenticate themselves with the previous session key. Since a device that is under physical attack has to be absent for at least one period, it will miss this period's session key and thus be unable to obtain any further keys. To prevent collusion between compromised devices, session keys are stored in lightweight secure hardware and are transmitted encrypted via secure channels. During the actual attestation, devices that fail to authenticate with the newest session key are regarded as physically compromised, whereas devices with a compromised software are detected based on existing software attestation techniques. In case of network partitioning or outages of leader devices, new leader devices are determined in the network, which take over the task of previous leaders. We show the security of our protocol against an adversary who compromises all but one device in the network and evaluate its scalability and robustness. Our results demonstrate that our protocol is highly efficient in well-connected networks and operates robust in very dynamic network topologies.

As a result, SCAPI provides several improvements over DARPA [68]. First, it reduces the number of transmitted messages per time period from $O(n^2)$ to $O(n)$, where n denotes the total number of devices in the network[1]. In this way,

1 In fact, when detecting physically compromised devices through their absence, $O(n)$ transmitted messages per time period is the best possible solution, since each device must at least send or receive one message.

SCAPI achieves scalability to millions of devices. Second, SCAPI is more robust against network delays by relying on a unidirectional 1-to-n delay-tolerant link in each session period, in contrast to an n-to-n continuous link. In addition, SCAPI can recover from device outages and network partitioning, which increases robustness against failures or targeted DoS attacks. Moreover, we present an extended version of SCAPI that further enhances delay robustness by requiring devices to contact their neighboring devices only once, at *any* time of the session period. Finally, SCAPI can precisely identify devices whose hardware and/or software is compromised, if less than half of all devices in the network are compromised.

Outline. In Section 6.2, we present SCAPI, our scalable attestation protocol to detect software and physical attacks. In Section 6.3, we extend the protocol to improve its robustness in dynamic network topologies. In Section 6.4, we evaluate the performance of SCAPI. Section 6.5 concludes this chapter.

6.2 SCAPI: SCALABLE ATTESTATION OF SOFTWARE AND PHYSICAL ATTACKS

SCAPI consists of three different phases. In the *deployment phase* (Section 6.2.1), the trusted network operator $\mathcal{O}$ initializes each device once, before the first operation of the network. The *session key update phase* (Section 6.2.2) is periodically executed during the operation of the network. In this phase, all physically uncompromised devices maintain a secret group key, the session key. We will show how the session key is periodically regenerated and propagated in the network. In addition, we will demonstrate that physically compromised devices are unable to obtain a valid session key. In the *attestation phase* (Section 6.2.3), $\mathcal{O}$ verifies the software and hardware integrity of all devices in the network and obtains a report that exhibits either the overall or the precise network state. As opposed to all our attestation protocols presented in previous chapters, SCAPI is secure against a physical adversary Adv_{hw} (Section 2.4).

6.2.1 *Deployment Phase*

Preliminaries. We discretize time into non-overlapping periods $t \in \{1, 2, 3, ...\}$ of a fixed session length δ. To protect against physical attacks of length δ_a (Section 2.4), we require $\delta \leq \delta_a/2$. The starting times of each time period are referenced with $P_1, P_2, P_3, ...$ The real time can be read by any device from a reliable read only clock (Section 2.3) using Time(), which for simplicity is assumed to be synchronized between all devices. Nevertheless, clock drifts could be tolerated by periodically synchronizing device clocks and considering the maximum allowed clock skew in δ_a. Moreover, each device keeps track of the time period t, running from time P_t until P_{t+1}, in which the device needs to update its session key.

Acronym	*Usage*
δ	length of time/session period
t	time pointer for next session key update
i	unique device identifier of $\mathcal{D}_i$
sk_{cur}	session key valid in current time period
sk_{next}	session key valid in next time period
ck_{ij}	channel key between $\mathcal{D}_i$ and $\mathcal{D}_j$
ak_i	attestation key to generate individual attests
dk_i	device key for authentication with $\mathcal{O}$

Table 6.1: Overview of secrets stored in the TEE of a device $\mathcal{D}_i$.

Enrollment. In the enrollment phase, the network operator $\mathcal{O}$ initializes the TEE (Section 2.3) of all devices with several secrets. First, devices store two initial session keys sk_{cur} and sk_{next}, which function as a group secret between all healthy devices. Second, each device $\mathcal{D}_i$ is equipped with two device-dependent symmetric keys dk_i, used during attestation to verify the authenticity of $\mathcal{O}$, and ak_i, used to generate device attests. In addition, each device stores t, a time pointer to the next session key update period, and its own unique device identifier i. Furthermore, $\mathcal{O}$ equips each device with the functionality to perform the session key update (Section 6.2.2) and attestation phase (Section 6.2.3). Any protocol code, except for network message processing, is stored and executed inside the TEE. For convenience, we assume an initial enrollment of all devices. However, $\mathcal{O}$ may enroll a new device at any point in time by first obtaining sk_{cur} and sk_{next} from the network and then initializing the new device as described above. Table 6.1 provides a summary of relevant definitions.

6.2.2 *Session Key Update Phase*

Basic Idea. The session key update phase is the core of SCAPI. It excludes devices from the network that are offline for an entire time period and, hence, are assumed to be physically compromised. For this purpose, a session key sk_{cur}, shared by all healthy devices, is periodically updated in a way that physically compromised devices are unable to obtain the newest sk_{cur}. In every time period, a so-called *leader device* generates a new secret session key sk_{next} for the subsequent time period, which is propagated in the network. Propagating sk_{next} requires sender and receiver device to authenticate themselves with the current session key sk_{cur}. At the start of every new time period, devices overwrite sk_{cur} with sk_{next} and the device leader generates and distributes a new sk_{next}.

A device that has been physically tampered with is offline for $\geq \delta_a$ and consequently misses the transmission of sk_{next} in at least one time period t_a, as

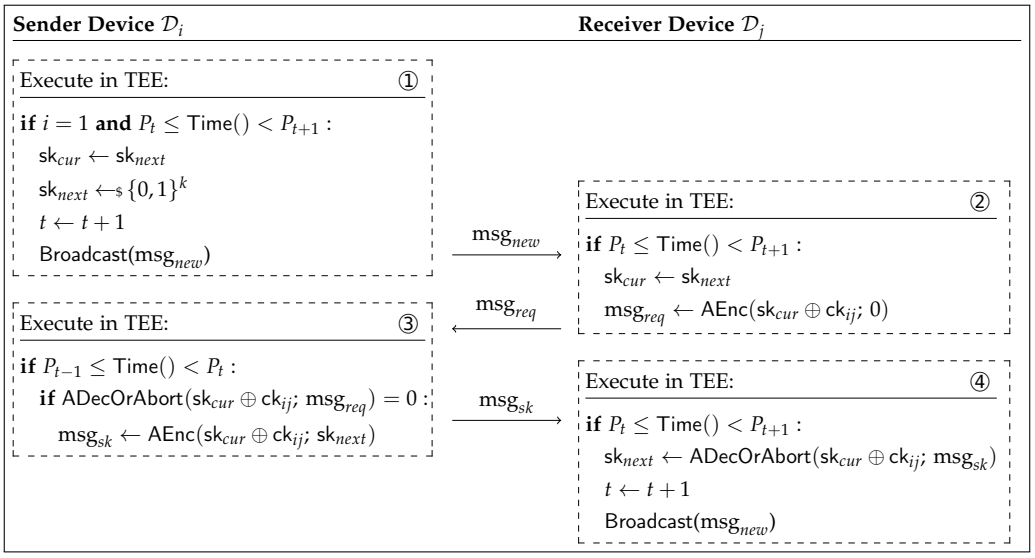


Figure 6.1: Session key transmission protocol between a sender $\mathcal{D}_i$ and a receiver $\mathcal{D}_j$ after channel key establishment, i.e., both devices share a channel key ck_{ij} and know their identities.

$\delta_a \geq 2\delta$. As a result, the compromised device does not possess a valid sk_{cur} for $t_a + 1$ and is also unable to obtain a valid session key for any subsequent time period $t_a + 2, t_a + 3, \ldots$. Thus, even when compromising many devices, Adv_{hw} is only able to obtain session keys of past time periods, which are of no use, as their validity already expired. Since SCAPI secures any communication using the current session key sk_{cur}, physically compromised devices are effectively excluded from the network, in particular during the execution of the attestation phase (Section 6.2.3). In the following, we describe the session key transmission protocol, formalized in Figure 6.1, in detail. It is run between two neighboring devices to transfer the session key from one device to the other.

Session Key Transmission Protocol. For simplicity, we henceforth assume that two neighboring devices have already established a unique symmetric channel key ck_{ij} that is stored in their TEEs. Pairwise channel keys between devices can either be preinstalled in the enrollment phase or established on demand when devices communicate with each other for the first time. In the latter case, devices perform a key exchange that is authenticated with the current session key sk_{cur} to prevent man-in-the-middle attacks [73].

The emission of the new session key in every time period is initialized by the leader device $\mathcal{D}_1$, which is the first device in the network (see ① in Figure 6.1). As soon as the leader observes that the real time Time() has reached the start of a new time period ($P_t \leq \mathsf{Time}() < P_{t+1}$), the leader first updates the session key of the current time period sk_{cur} to the most recently exchanged key sk_{next}. We remark that there is no need to store past keys, which could be denoted as sk_1,sk_2, $\mathsf{sk}_3, \ldots$, since only the current and next session key are relevant for any device. After updating the current session key, the leader generates a new

random session key sk_{next} for the subsequent period $t+1$ and increments its time pointer t by one. Consequently, the time period described by the pointer is now ahead of the real time $P_t > \mathsf{Time}()$. This indicates that a device is in possession of a valid session key for the upcoming time period. Next, the leader informs its neighbors about the new session key with a message msg_{new}.

On receiving msg_{new} from any device $\mathcal{D}_i$, a device $\mathcal{D}_j$ will enter its TEE and check whether the next time period has been reached (see ②). If this is the case, $\mathcal{D}_j$ will update its current session key to the previously communicated one. Afterwards, $\mathcal{D}_j$ encrypts and authenticates ($\mathsf{AEnc}()$) a fixed string, e.g., '0' with the current session key sk_{cur} that is XOR-ed with the channel key ck_{ij} shared by both devices. The result is sent from $\mathcal{D}_j$ to $\mathcal{D}_i$ in msg_{req}.

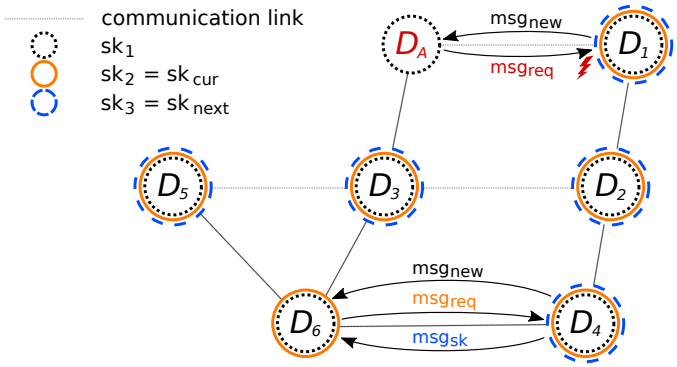
Next, $\mathcal{D}_i$ verifies the correctness of msg_{req} from $\mathcal{D}_j$ (see ③). A successful authenticated decryption ($\mathsf{ADecOrAbort}()$) shows $\mathcal{D}_i$ that $\mathcal{D}_j$ is in possession of the current session key, and thus physically uncompromised. Only then, $\mathcal{D}_i$ answers with a message msg_{sk}, containing the next session key sk_{next}, which is also authenticated and encrypted with sk_{cur} XOR-ed with ck_{ij}. Message msg_{sk} is a good example to emphasize the necessity of the channel key ck_{ij}. In this case, ck_{ij}, only known to $\mathcal{D}_i$ and $\mathcal{D}_j$, prevents Adv_{hw} from obtaining the newest session key. If ck_{ij} is not used to encrypt msg_{sk}, Adv_{hw} could gradually decrypt recorded msg_{sk} messages with an old (expired) session key that Adv_{hw} obtained from a third physically compromised $\mathcal{D}_A$, and finally decrypt and obtain the newest session key.

On receiving msg_{sk}, $\mathcal{D}_j$ performs an authenticated decryption of msg_{sk}. If it succeeds, $\mathcal{D}_j$ stores the new session key as sk_{next}, increments its time period pointer, and announces the new session key to its neighbors with msg_{new} (see ④). Neighboring devices of $\mathcal{D}_j$ that have not yet received the newest session key then perform the session key transmission protocol with $\mathcal{D}_j$. This time $\mathcal{D}_j$ acts as the sender and the particular neighboring device as the receiver. Figure 6.2a illustrates the session update phase in a network with 6 healthy devices and one adversarial device $\mathcal{D}_A$ that was physically compromised in time period $t = 2$.

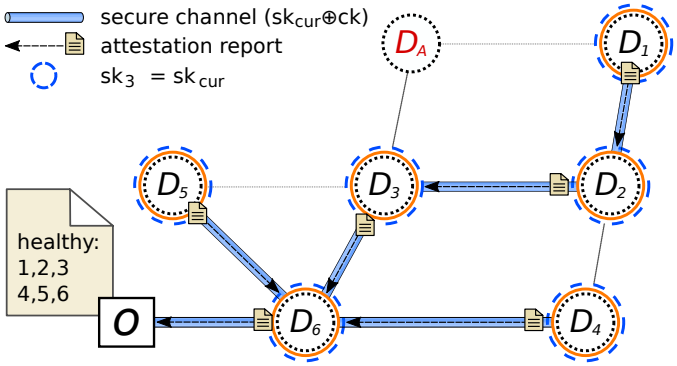
We note that the update of the session key relies on the availability of the leader device, which constitutes a single point of failure. In Section 6.3, we extend the protocol to improve its robustness against device outages, network delays and partitioning, or targeted DoS attacks. Furthermore, this as well as all further SCAPI protocols are presented in a simplified way for reasons of clarification. In practice, message types must be authenticated and timeouts must prevent devices from hanging during protocol execution upon errors.

6.2.3 *Attestation Phase*

Basic Idea. The attestation phase allows the network operator $\mathcal{O}$ to verify the integrity of the software and hardware of all network devices. To initiate the attestation phase, $\mathcal{O}$ connects to an arbitrary device and sends an attestation request, which is then propagated in the network and answered by all devices



(a) *Session key update phase:* All devices store the initial session key sk_1 that was used to secure the exchange of sk_2. Subsequently, sk_2 is used to exchange the next session key sk_3 for the upcoming time period $t = 3$. Such an exchange is illustrated between $\mathcal{D}_4$ and $\mathcal{D}_6$. We observe that $\mathcal{D}_A$ was physically compromised in time period $t = 2$ and thus did not receive sk_2. Consequently, $\mathcal{D}_A$ is unable to obtain sk_3 or any following session key.



(b) *Attestation phase:* The attestation request from $\mathcal{O}$ was forwarded to all devices that were in possession of the current session key sk_3. $\mathcal{D}_A$'s attest is not included in the attestation report, as $\mathcal{D}_A$ is excluded from all communication in the attestation protocol. Using a spanning tree topology, attestation reports are propagated back to $\mathcal{O}$ and aggregated at each hop.

Figure 6.2: In Figure 6.2a a session key update phase is illustrated for 7 devices in time period $t = 2$. In Figure 6.2b, the same network is illustrated in time period $t = 3$ while answering an attestation request from $\mathcal{O}$.

with an attestation report. Emitting an attestation request, $\mathcal{O}$ selects to verify either the *overall* or the *precise network state*. The former is secure against an adversary who compromises all but one device ($\lambda = 1$, see Section 2.4), but only outputs a Boolean result, namely whether all devices are healthy or not. The latter precisely identifies compromised devices by id. Yet, it relies on the assumption that at least half of all devices in the network are healthy, i.e., $\lambda \geq n/2$, with n being the total number of prover devices. Propagating the attestation request through the network arranges a spanning tree whose root is $\mathcal{O}$. This enables efficient transmission and aggregation of attestation reports along the spanning tree back to $\mathcal{O}$ [7, 11, 68].

Instead of relying on a tree-based approach, SCAPI can also apply the distributed approach of SALAD (Chapter 5). Building on SALAD enables SCAPI to

perform the attestation phase in highly dynamic and disruptive networks, albeit at the cost of scalability and efficiency. To this end, only a minor modification to the attestation phase of SALAD is necessary: Any communication between two devices $\mathcal{D}_i$ and $\mathcal{D}_j$ must be authenticated and encrypted using the current session key sk_{cur} XOR-ed with the channel key ck_{ij} shared by both devices. Since physically compromised devices lack sk_{cur}, this prevents Adv_{hw} from participating in the attestation phase, so that $\mathcal{O}$ receives a report that only contains healthy devices.

In the following, we first introduce essential building blocks, namely, the verification of the software integrity (VerifySW), and the generation (GenRep), aggregation (MergeRep), and verification (VerRep) of attestation reports. Although function names for handling attestation reports are the same in SCAPI and SALAD (Chapter 5), their implementations are different. This is because SCAPI uses a tree-based aggregation approach, whereas SALAD relies on a distributed approach. After introducing all building blocks, we describe the attestation protocol, which is formalized in Figure 6.3. Figure 6.2b illustrates the attestation phase in a network with 6 healthy devices and one adversary device $\mathcal{D}_A$ that was physically compromised.

Verification of Software Integrity. When initiating an attestation of the network, $\mathcal{O}$ defines a set of valid software states *vss* in the attestation request. *Vss* specifies all software configurations that are permitted by $\mathcal{O}$, e.g., because they represent the correct and most recent software states. Devices implement a function VerifySW() in their TEE that takes *vss* as an argument. Internally, VerifySW() measures the local software configuration (*vss* could contain a description what should be measured) and generates a measurement that is compared to the respective expected measurement value defined by $\mathcal{O}$ in *vss*. VerifySW() returns *true* if both values match, and *false* otherwise. We deliberately abstract from any implementation details for the underlying integrity measurement mechanism to support a wide range of attestation mechanisms.

During the execution of the attestation protocol, each device determines its software state. Devices being in an untrustworthy software state (VerifySW(*vss*) = *false*) immediately abort the attestation phase, such that $\mathcal{O}$ receives a report that exclusively contains healthy devices.

Attestation Reports. During attestation, each healthy device generates an *attest*, which proves that the device is in a healthy software and hardware state. A device $\mathcal{D}_i$ generates its attest by computing an HMAC with its attestation key ak_i over an initial challenge *ts* from $\mathcal{O}$. An *attestation report* condenses attests of one or multiple devices. It consists of a secure *aggregate* and, if $\mathcal{O}$ verifies the precise network state, a *device description*. The aggregate contains the attests of all devices that contributed to the attestation report. A secure aggregate of multiple attests is computed by XOR-ing all attests [77]. The device description lists all devices whose attest is in the aggregate and is only required when $\mathcal{O}$ requests for the precise network state. To minimize the size of the device description, it can be represented in three different ways: as a list of device ids present in the

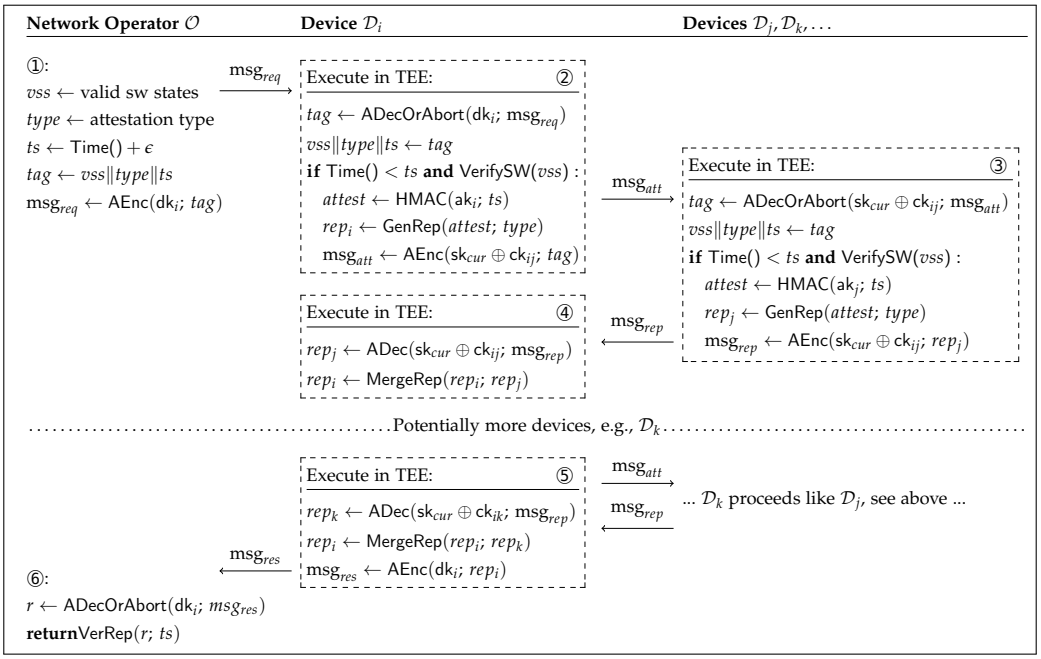


Figure 6.3: Illustration of the attestation protocol after secure channel establishment, i.e., all devices share a channel key ck and know their identities.

aggregate, as a list of ids not present in the aggregate, or as an n-bit vector where a one at position k indicates that $\mathcal{D}_k$ is in the aggregate. An attestation report is generated by the function GenRep(), which takes as input the attest of a device and the attestation type (overall or precise state). Multiple attestation reports are aggregated by the function MergeRep(), which merges all device descriptions (if available) and XORs all attests.

$\mathcal{O}$ is able to verify an attestation report by passing the report and the initial challenge ts to the function VerRep(). It recomputes the attests for all devices, whose ids are contained in the device description. Given no description, $\mathcal{O}$ recomputes the attests for all devices. If the recomputed aggregate equals the reported aggregate and if at least $n/2$ attests are included in the report, then the report is assumed to be valid. Only then, all attested devices are assumed to be healthy and VerRep() returns a bit vector, where a zero/one at position k indicates that $\mathcal{D}_k$ is in a compromised/healthy state.

Attestation Protocol. Initially, the network operator $\mathcal{O}$ connects to a device $\mathcal{D}_i$ and emits an attestation request msg_{req} (see ① in Figure 6.3). The request is authenticated and encrypted with the device key dk_i, known only to $\mathcal{D}_i$ and $\mathcal{O}$, and contains all valid software states vss that are permitted by $\mathcal{O}$. In addition, the request comprises the attestation $type$, i.e., overall or precise network attestation, and a timestamp ts that is dated in the future and serves two purposes: (1) as an expiry date ($\mathsf{Time}() + \epsilon$) for the attestation request, and (2) as a challenge for the computation of individual device attests.

Next, $\mathcal{D}_i$ verifies the authenticity and freshness of the attestation request to prevent Adv_{hw} from performing network-wide denial of service attacks (see ②). Furthermore, $\mathcal{D}_i$ verifies whether it is in a trustworthy software state by passing *vss* to VerifySW(). If all checks pass, $\mathcal{D}_i$ generates its individual attestation report rep_i, which initially only contains $\mathcal{D}_i$'s own attest. Moreover, $\mathcal{D}_i$ propagates the attestation request to all neighboring devices. This and all following communication between devices is secured with the current session key sk_{cur} and the respective channel key ck, in order to prevent physically compromised devices from participating in the attestation protocol.

Any device that receives an attestation request first verifies the request and then also propagates the request to its neighboring devices. These steps are repeated until the attestation request reaches leaf devices whose neighbors already have received a request. Leaf devices return their attestation report to their parent device, i.e., the device from which they initially obtained the attestation request (see ③). Parent devices merge their own attestation report with all reports they receive from child devices (see ④), and propagate the merged report to their parent devices. Eventually, $\mathcal{D}_i$ merges a final report that contains all healthy devices in the network. This final report is encrypted under dk_i and transmitted to $\mathcal{O}$ (see ⑤).

$\mathcal{O}$ verifies the report by passing it with the initial challenge *ts* to VerRep() (see ⑥). If the report is valid, $\mathcal{O}$ assumes that all devices listed in the report are healthy. Physically compromised devices could not participate in the protocol, as they lacked sk_{cur}. Likewise, devices with a compromised software did not contribute to the attestation report, since they aborted protocol execution.

We note that if the attestation phase is executed while the session key is updated, some parent devices may already have the new session key sk_{next} while their child devices still possess the old sk_{cur}. To prevent synchronization issues, such parent devices must first propagate sk_{next} to their child devices, before they receive the encrypted attestation reports from them. Additionally, the attestation protocol must either complete in time δ_a or $\mathcal{O}$ has to periodically check the presence of $\mathcal{D}_i$ during attestation. In the latter case, there is no restriction for the attestation phase runtime. The measures are necessary to prevent Adv_{hw} from physically compromising $\mathcal{D}_i$ during the attestation phase, which would enable Adv_{hw} to extract an aggregate and induce attests of physically compromised devices. Furthermore, the attestation phase currently does not tolerate device mobility or device outages during protocol execution. For the purpose of handling low to medium network dynamics and disruptions, SCAPI can be extended with common ad-hoc networking techniques, i.e., a routing protocol, introducing timeouts, and maintaining a virtual spanning tree topology. In order to handle high network dynamics and disruptions, SALAD's distributed approach can be applied to SCAPI, as previously described.

6.3 ROBUSTNESS EXTENSION

In wireless networks, device mobility can lead to broken connections, which increase communication delays or even cause temporarily partitioned networks. To tolerate network dynamics in the attestation phase, either a virtual spanning tree or SALAD's distributed approach can be employed (see Section 6.2.3). In this way, the functionality and security of the attestation protocol is preserved, albeit, communication and runtime increases. However, network disruptions are much more severe during the session key update phase, since increased delays or temporary outages can provoke false positive errors, where healthy devices are mistakenly regarded as physically compromised. This is a common issue that all absence detection approaches have to contend with [32, 33, 62, 65, 68]. The issue is caused by their underlying assumption that devices being absent for more than a certain amount of time δ are regarded as physically compromised.

In this section, we extend the session key update phase (Section 6.2.2) to minimize the amount of false positives during attestation. Thereby, we increase SCAPI's *robustness* against network partitioning, failures, and delays, making it more suitable for dynamic and disruptive network topologies.

Part 1 – Tolerating Device Outages. When attesting the precise network state, SCAPI tolerates up to $n/2$ device outages with the exception of the leader device. This is an improvement compared to DARPA [68], which is intolerant towards a single device outage. Nevertheless, the leader device constitutes a single point of failure that can render the protocol dysfunctional, e.g., upon a leader device outage, network partitioning, or targeted DoS attacks. To continue service when the leader itself becomes absent, we propose that devices in the network elect a new leader in a self-organizing manner. The new leader is identified by the lowest device identifier and takes over the task of the previous leader, i.e., the periodic emission of a new session key.

In detail, devices now store the device identifier (*id*) of their current leader in $\mathcal{D}_{min}$, which is set to 1 during enrollment. When a device fails to receive the next session key from the current leader within a time δ_{out} ($\delta_{out} \ll \delta$), it first elects itself as a temporary leader. A temporary leader initially generates its own next session key, sets $\mathcal{D}_{min}$ to its own *id*, and periodically announces the existence of a new session key with msg_{new}. For efficiency reasons, we combine this leader election with the transmission of the next session key, issued by the new leader. Therefore, msg_{new}, msg_{req}, and msg_{sk} now additionally carry the *id* of the device whose session key is announced, requested, or transmitted. A device $\mathcal{D}_j$ that receives msg_{new} from a neighboring device $\mathcal{D}_i$, compares the announced *id* with its stored $\mathcal{D}_{min}$. In case *id* is lower, $\mathcal{D}_j$ performs the session key transmission protocol (Section 6.2.2) with $\mathcal{D}_i$ to obtain the next session key of device *id*. On success, $\mathcal{D}_j$ sets its $\mathcal{D}_{min}$ to *id*, regards $\mathcal{D}_{id}$ as leader, and henceforth announces and distributes the next session key of $\mathcal{D}_{id}$. In this manner, the *id* and next session key of the device with the lowest *id* will propagate in the network and will replace all other temporary leaders as well as their session keys. The newly

elected leader recognizes itself as the leader, when it only received messages from devices with higher *id*s during election. Note that the original leader has the smallest *id* in the entire network, hence, its return is implicitly tolerated within the same session.

Part 2 – Tolerating Link Delays. Thus far, the session key update phase relies on a 1-to-n delay-tolerant link between the possibly changing leader and all other devices. Although this is a major advance over DARPA [68], which requires a continuous n-to-n link, in certain network topologies the leader may not be able to continually reach all devices within time δ. In general, δ can be increased to meet the dynamics of such networks. Yet, this comes at the expense of an increased attacking time $\delta_a \geq 2 \cdot \delta$, hence, lower security. In the following, we present an alternative solution to increase robustness against extreme delays. In our approach, devices support multiple leaders and session keys per period by propagating not only the leader's next session key, but k next session keys from devices with the lowest *id* in the network. Thus, up to k ($\leq n$), instead of 1, session keys can be valid in every time period. As a result, SCAPI is robust and secure as long as the network can be represented as an *undirected connected graph*, where (i) each node depicts a device, and (ii) an edge between two nodes indicates that both devices share at least one common session key.

In detail, devices that fail to receive sk_{next} from their leader generate their own next session key and become leaders themselves, as described above. However, instead of forwarding only a single key from their future leader, devices now store and exchange multiple current and next session keys. For each current and next session key devices store, they record the identifier *id* of the device that generated the key. This list of stored key *ids* is appended to each emitted msg_{new}. Nevertheless, to reduce communication overhead, we propose that devices limit the number of advertised key *ids* in msg_{new} to k entries, namely, those with the lowest *id*. A device $\mathcal{D}_j$ that receives msg_{new} from a neighboring device $\mathcal{D}_i$ compares the advertised key *id* list with its own stored keys. If $\mathcal{D}_i$ advertised the *id* of a current session key sk_{cur} that $\mathcal{D}_j$ stores, both devices can continue executing the key transmission protocol and use sk_{cur} as a session key to secure communication (see Section 6.2.2). However, they only do so, if $\mathcal{D}_i$ also advertised at least one key that is new to $\mathcal{D}_j$, i.e., $\mathcal{D}_i$ listed an *id* in msg_{new} that is not contained in $\mathcal{D}_j$'s stored key *id* list. If this is the case, $\mathcal{D}_j$ requests all such new keys from $\mathcal{D}_i$ by listing their *ids* in msg_{req}. Receiving msg_{req} with a list of requested current and/or next key *ids*, $\mathcal{D}_i$ answers with a msg_{sk} that contains the actual value of the requested keys.

Note that devices exchange the k-th *current and next* session keys from devices with the lowest *id*. This reduces fragmentation in the *current and next* session period, i.e., the number of devices not sharing a common session key. A higher k provides more robustness, but increases the communication overhead in the network. If $k = 1$, the approach corresponds to the extension in part one.

Security. All protocol extension code and data is stored in the TEE of devices, and device identifiers as well as session keys in msg_{req} and msg_{sk} are transmitted

authenticated and encrypted by $\mathsf{sk}_{cur} \oplus \mathsf{ck}$, with sk_{cur} being issued by any previous leader device. The key *ids* in msg_{new} can be transmitted in plain, as any forgery will inevitable be detected by the sender when processing msg_{req}. Therefore, the security of the proposed extension can be reduced to the security of the original session key update phase (Section 6.2.2).

6.4 EVALUATION

In the following, we evaluate SCAPI and its protocol extensions. First, we describe our setup, give details of the implementation, and present our measurements (Section 6.4.1). Then, we report on simulation results conducted in static large-scale networks to demonstrate the scalability of SCAPI (Section 6.4.2) as well as dynamic network topologies to show the robustness of our protocol (Section 6.4.3).

6.4.1 *Implementation and Measurements*

Setup. We used the same setup as in our secure code update scheme (Section 4.2) and SALAD (Section 5.4). Accordingly, we implemented our protocol on Stellaris EK-LM4F120XL microcontrollers that were equipped with Anaren's CC2530 BoosterPacks. The Stellaris provides the minimal hardware requirements to perform remote attestation [127], but unfortunately lacks a write-protected device clock to prevent malware from tampering with the device time (Section 2.3).

Runtime Measurements. We based our implementation on the same libraries that we used for our secure code update scheme. Please refer to Section 4.2.3 for network and cryptographic runtime measurements.

Memory Costs. In our implementation, devices store their own ECDH key pair (64 bytes), the current and the next session key (each 16 bytes), the leader device id (4 bytes), j secure channel keys and device ids (each 20 bytes), and a timestamp (4 bytes). The number j of stored secure channel keys can be adjusted to the particular memory requirements, since devices can establish channel keys right away by performing an ECDH key exchange with a neighboring device. If the multiple leader protocol extension is used, devices store up to k current and next session keys and their respective device id (each 20 bytes). Further, devices need to temporarily store the attestation report during attestation. The size of the attestation report is dependent on the number n of devices and the number of attests contained in the report. In the worst case, where the device description must be represented as an n-bit vector, it amounts to $n/8 + 64$ bytes. However, using another representation, such as a list of device ids contained in the report (Section 6.2.3), the actual size of the report is much lower for many devices in the network. In total, devices require $72 + j \cdot 20 + k \cdot 40$ bytes of permanent storage and at most $n/8 + 64$ bytes of temporary storage.

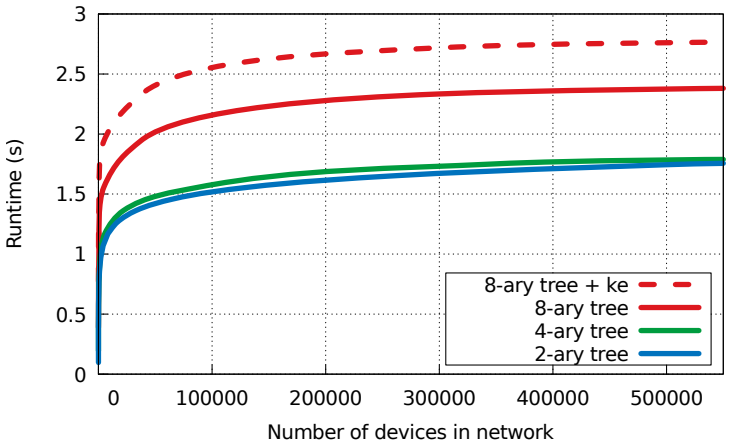


Figure 6.4: Session key update runtimes in various static topologies. The dotted line includes the initially needed key exchanges (ke).

6.4.2 *Scalability Simulation Results*

Setup. In order to demonstrate the scalability of SCAPI, we performed network simulations in static network topologies. In these networks, all devices are connected and stationary, so there are no link breaks or delays for determining routes in the network. We used the OMNeT++ framework [146] to simulate a homogeneous network with ten to multiple million Stellaris devices, implemented our protocol on the application layer, and used computational and network delays based on our measurements.

Session Key Update Runtime. We simulated the runtime of the session key update phase in various topologies. Figure 6.4 shows the runtime for a binary, 4-ary, and 8-ary tree topology with up to 550 000 devices, where the leader device is located at the root of the tree. The figure demonstrates that in tree network topologies, runtime increases logarithmically with the number of devices in the network. Under these conditions, the transmission of next session keys achieves an outstanding performance, requiring less than 2.4 seconds to reach 500 000 devices in an 8-ary-tree and less than 1.8 seconds in a binary tree topology. Even with multiple million devices, runtime remains below 2 seconds in a binary tree topology. Only the first propagation of the next session key in the network requires little more time, since neighboring devices initially need to exchange public keys and perform key exchanges to establish their channel keys. Yet, even with the additional key exchanges, runtime remains below 2.8 seconds for more than 500 000 devices.

Attestation Runtime. Figure 6.5 shows the runtime of the attestation phase for a binary and 8-ary tree topology with up to 550 000 devices, where the network operator is located at the root of the tree. During attestation, devices verify the integrity of installed software by computing a SHA512 digest over a 30 kB software and comparing the digest to an expected value that is specified in the

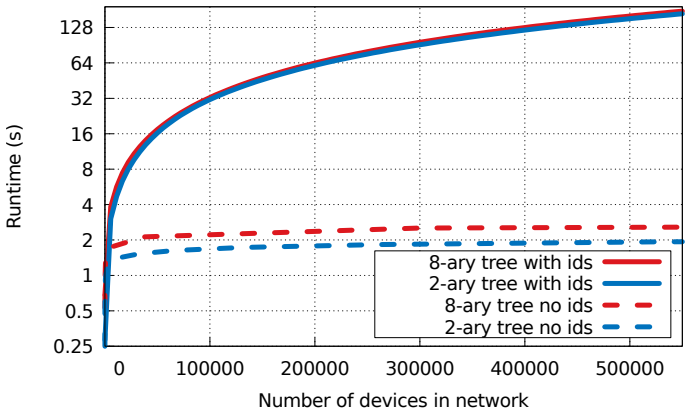


Figure 6.5: Attestation runtimes in two static topologies for verifying the overall (no ids) or precise (with ids) network state.

attestation request. Additionally, we varied the attestation type, requesting either for the precise network state (solid lines) or the overall network state (dashed lines). Figure 6.5 demonstrates that reporting precise device identifier introduces a notable overhead. When reporting the overall network state, attestation runtime increases barely with the number of devices in the network, remaining below 3 seconds even for networks with multiple million devices in almost any tree topology. Yet, when reporting precise device ids, runtime increases to 158 seconds for 500 000 devices due to the large size of the attestation report, which increases proportionally with the network size. Nevertheless, we consider that 2.5 minutes is an acceptable timeframe to obtain a report that precisely lists which devices out of half a million are in a compromised state. Moreover, to increase efficiency, $\mathcal{O}$ may only verify the precise network state, if the (faster) verification of the overall network state fails.

Communication Costs. During a session key update phase, each device sends and receives multiple msg_{new} (1 byte), msg_{req} (45 bytes), and msg_{sk} (45 bytes) messages, depending on the particular network topology and the position of the respective device in that topology (i.e., root, middle, or leaf device). If devices need to establish a secure channel key, they need to mutually exchange their public keys, which causes an additional message overhead of 32 bytes. For instance, in a binary tree topology, devices transmit on average 182 bytes, or 310 bytes with the initial key exchange, in each session key update phase.

During the attestation phase, devices receive one msg_{req} or msg_{att} (both 108 bytes), send a msg_{att} and receive a msg_{rep} to/from non-leaf neighbor devices, and send one msg_{rep} or msg_{res} (both $\leq n/8 + 92$ bytes for precise state or 92 bytes for overall state) to their parent device or $\mathcal{O}$. In summary, assuming a binary tree topology, devices transmit on average 400 bytes, if the overall network state is attested. If the precise network state is attested and the network consists of $n = 10\,000$ devices in a binary tree topology, devices transmit on average 1314 bytes.

Summary. We demonstrated that SCAPI is highly efficient in static network topologies. In comparison to DARPA [68], we reduce the number of transmitted messages per session period from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$, thus, decreasing protocol communication costs and runtime. To illustrate this advantage, in binary-tree topologies our approach is 27 times faster with $n = 2\,000$ devices and 3800 times faster with 500 000 devices. As a result, SCAPI saves energy costs, supports larger networks, and tolerates a substantially shorter session length δ (e.g., a 3 seconds δ is sufficient in the presented network topologies). Note that a shorter session length increases security, as $\delta_a \geq 2 \cdot \delta$. Furthermore, for comparison we already considered the fastest variant presented in [68], which requires each device to store n symmetric keys. In our protocol, devices must only store the keys of neighboring devices, e.g., 3 in a binary tree topology.

When attesting the state of the entire network, both protocols ([68] and SCAPI) show a runtime that scales logarithmically with n. Nevertheless, in contrast to [68], SCAPI also allows to determine the ids of compromised devices with an acceptable overhead even in larger networks.

6.4.3 *Robustness Simulation Results*

Setup. We evaluated the robustness of SCAPI to false positives by simulating worst-case scenarios, where the network topology is highly partitioned as well as constantly changing. To this end, we randomly deployed devices in a 1000m×1000m square area and applied a random waypoint mobility model, where each device moves at a random speed between 5 and 15 m/s. In order to simulate link disruptions, network delays, and signal interference in a realistic manner, we modeled a 802.15.4 physical and medium access control layer using the INET framework for OMNET++ [64]. We set the wireless communication range to 50m, representing 50% of the distance specified in the ZigBee standard, and the session period to 5 minutes, detecting physical attacks that require more than 10 minutes. Simulating both layers, device mobility, and multiple days of runtime requires a huge amount of computational power, making it infeasible to scale simulations to thousands of devices [21]. Nevertheless, as we will show in the following, the main hurdle is to operate robust in sparse topologies, where the network is constantly partitioned. We showed in the previous section that our protocol performs well in dense networks with permanently connected devices.

Robustness. We examined the elapsed time until the absence detection of SCAPI produces false positives, i.e., healthy devices that are regarded as physically compromised because they did not receive the next session key on time in the key update phase. Figure 6.6 illustrates the average runtime until a certain amount of false positives occur with or without the robustness extension (Section 6.3). The figure shows that the network must be sufficiently dense in order that the randomly moving devices meet each other frequently enough to exchange the next session key on time. In fact, there is an exponential correlation between

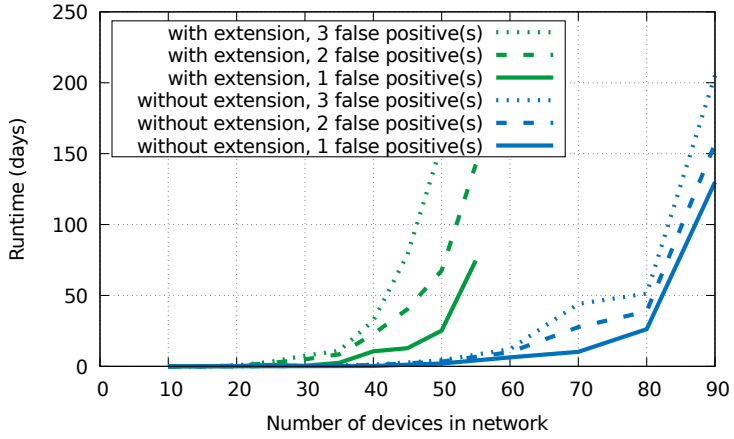


Figure 6.6: Average runtime of SCAPI in highly dynamic networks until false positives occur.

device density and robustness. This causes the average error-free runtime to quickly increase from 2.5 days with 35 devices to 72.8 days with 55 devices, when using the robustness extension. To illustrate the sparseness in this scenario, 35 optimally distributed and connected devices cover 17.0% of the area and 55 devices 26.6%. The figure also demonstrates the effectiveness of the robustness extension. Without extension, approximately 60% more devices are required to achieve comparable error-free runtimes. Investigating false positives, we identified the main cause in the random movement of devices. Commonly, a single device does not encounter other devices, and thus has no chance to receive the next session key on time. This cannot be prevented by faster computations or smaller communication delays in our protocol, but only by increasing the duration of the session period. We observed that this hiding of single devices has barely any cascading effect. Hence, as shown in Figure 6.6, if tolerating a minimal amount of false positives, significantly longer runtimes are possible.

We further analyzed the time required in each session key update phase to establish an error-free network state with and without the robustness extension. Figure 6.7 shows for a varying amount of time the largest fraction of devices that shares a common next session key (without extension) or is connected through multiple next session keys (with extension). The figure illustrates the importance of the network density. In dense networks, session keys can spread faster and thus reach more devices in shorter time. Nevertheless, with our proposed extension, we eliminate the bottleneck that the leader device's next session key must reach all other devices on time. Instead, temporarily unconnected devices generate and distribute their own individual next session keys and tolerate multiple valid keys per period. Hence, a robust network state is already established if devices exchanged their individual session keys with immediate neighboring devices. In the described setting and with 50 devices, this state is already reached within 182 seconds.

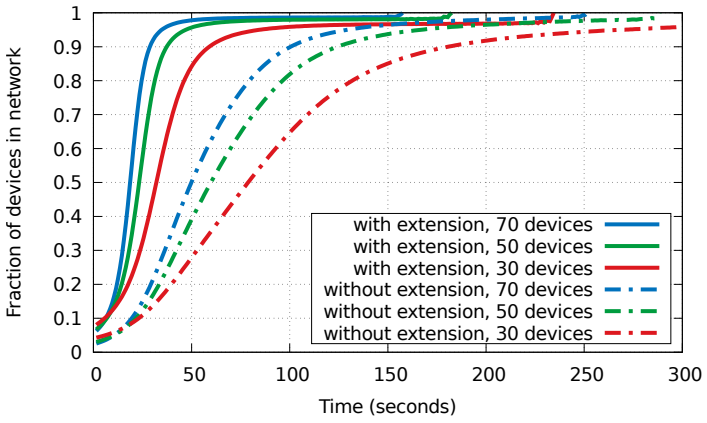


Figure 6.7: Average runtime required for the distribution of next session keys.

Summary. We demonstrated the robustness of SCAPI to false positives in extreme scenarios. Although our protocol targets connected network topologies, we showed that it also performs robust in disruptive and highly dynamic topologies, where it achieves error-free runtimes in the range of multiple weeks. We also exhibited the advantage of our protocol extension (Section 6.3). It provides the same robustness as the unextended protocol in almost half as dense networks. By contrast, DARPA [68] relies on continuous links, and hence is unable to deal with network disruptions. Yet, even if DARPA is extended to support disruptions, n devices must successfully deliver a message to n other devices in each session period. In comparison, SCAPI improves robustness in two stages. First, our unextended protocol relies on *one* device sending a message to n devices. Second, in the extended version, devices must merely reach immediate neighboring devices once, at *any* time during a session period.

6.5 SUMMARY

We presented SCAPI, a scalable attestation protocol that detects physical attacks. Compared to DARPA [68], our protocol reduces the number of transmitted messages per time period from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$, thus scaling to millions of devices and outperforming existing solutions by orders of magnitude. In addition to verifying the overall state of the network, SCAPI is able to precisely identify devices that run compromised software or have been physically manipulated. We demonstrated that our protocol is robust even in very dynamic network topologies, as it quickly recovers from device outages or network partitioning.

7 ATTESTATION OF AUTONOMOUS EMBEDDED SYSTEMS

In autonomous networks, embedded devices operate collaboratively in a distributed and self-organizing manner. These autonomous embedded systems have special requirements on scalability, performance, robustness, and security, which are only insufficiently addressed by existing attestation protocols. In this chapter, we propose PASTA, a practical attestation protocol for autonomous embedded systems. Compared with our previously proposed protocols, PASTA expands the 1:n relation between verifier, i.e., network operator, and prover devices to m:n. In particular, PASTA enables many low-end embedded prover devices to attest their integrity towards many potentially untrustworthy low-end embedded verifier devices. Furthermore, PASTA is fully decentralized and thus able to overcome arbitrary network and device outages. Like SCAPI (Chapter 6), PASTA can detect physical attacks, albeit in a more robust way, which suits the demands of autonomous embedded systems. We implement our protocol, conduct measurements, and simulate large networks. The simulation results show that our protocol scales to large networks with millions of devices and improves robustness by multiple orders of magnitude compared with SCAPI.

Remark. Parts of this chapter have been published in [84].

7.1 MOTIVATION AND CONTRIBUTION

Embedded devices are increasingly deployed in distributed and autonomous networks, in which devices operate collaboratively with minimal or no supervision. Specific application scenarios include industrial control [31], smart cities [74], building automation [118], logistics [72], or environmental monitoring [53]. However, existing attestation protocols are unable to fulfill the requirements on scalability, performance, robustness, and security that are needed for autonomous embedded systems. In the following, we elaborate on these requirements and provide a summary in Table 7.1.

Scalability and Performance: Autonomous networks of embedded systems may comprise thousands of collaborating low-end embedded devices. For instance, a network may contain many actuator devices that control a safety-critical process based on measurements received from many sensor devices. In this case, all actuators need to verify that all sensors from which they receive data are uncompromised. Hence, sensors must be provers and actuators must be verifiers (and potentially provers as well). Collective attestation protocols (see Section 2.1.2) address this issue to some extent, as they distribute the computational and communication burden across all provers in the network [7, 11, 28, 67, 68]. This allows a scalable and efficient attestation of the entire network. Yet, collective

	Scalability		*Performance*		*Robustness*		*Security*
	Scalable Prover Attestation	**Publicly Verifiable Report**	**Efficient Report Generation**	**Efficient Report Verification**	**Decentralized Attestation**	**Robust to Net. Disruptions**	**Detection of Physical Attacks**
SEDA [11]	●	○	●	●	○	◐	○
SANA [7]	●	●	◐	○	○	◐	○
DARPA [68]	●	○	●	●	○	○	●
SeED [67]	●	○	●	●	○	◐	○
LISA [28]	●	○	●	●	○	◐	○
SALAD (§ 5)	◐	○	●	●	○	●	○
SCAPI (§ 6)	●	○	●	●	○	◐	●
PASTA (§ 7)	●	●	●	●	●	●	●

Table 7.1: Overview of collective attestation protocols and features important for the attestation of autonomous embedded systems.

attestation protocols are typically centralized, meaning that only a single entity, which is usually the network operator, is capable of verifying the attestation result. In fact, only one collective attestation protocol considers multiple verifiers [7]. However, it poses a high computational burden on provers and even more on verifiers, which makes it unsuitable in cases where provers and especially verifiers perform time-critical operations or have limited computational power, e.g., as they are low-end embedded devices[1].

Robustness: Autonomous systems typically need to operate undisturbed without manual intervention for long times, during which the system may have to overcome network and device disruptions. Therefore, an attestation protocol for autonomous systems must be robust and sustain its security service in case of failures. However, the dependence of existing protocols on a centralized [11, 28, 67, 68] or external verifier [7] that initiates, controls, and verifies the attestation, constitutes a single point of failure, which impairs robustness.

Security: For reasons explained in previous chapters, embedded devices are often at a higher risk of being physically manipulated than general-purpose computers. Collective attestation protocols have recently been combined with absence detection (see Section 2.1.2) to also detect physical attacks. Unfortunately, existing attestation protocols that consider physical attacks are prone to network and device outages, whereupon healthy, but temporarily unreachable, provers are mistakenly regarded as physically compromised. Thus, their additionally limited robustness make them unsuitable for autonomous systems.

1 For example, the authors report that a 48 MHz embedded prover device requires more than 2.2 seconds to generate its attestation report and a server (Amazon EC2 t2.micro instance) requires more than 1 second to verify an attestation report containing 1000 prover devices [7].

Contribution. In this chapter, we propose PASTA, a practical attestation protocol for autonomous networks of embedded devices. In PASTA, prover devices periodically collaborate to generate so-called tokens. Each token attests the integrity of all provers that participated in its generation. During token generation, provers mutually ensure their integrity and then make use of a Schnorr-based multisignature scheme to compute an aggregated signature, which is stored in the token and attests the provers' integrity. The aggregated signature is of constant size and can be efficiently generated, which ensures scalability and performance. Furthermore, the aggregated signature is publicly and efficiently verifiable, which enables even untrustworthy low-end embedded devices to rapidly verify the integrity of all provers.

To increase the availability of the attestation result in case of network and device outages, tokens are distributed to all devices in the network, instead of being stored centralized. Since tokens are protected by their contained signature, any device can store and forward tokens, which facilitates their quick distribution. Moreover, PASTA is fully decentralized, because all provers equally ensure the freshness of tokens by periodically initiating a new token generation. Thus, in case of arbitrary network or device outages, all remaining operative and connected provers still sustain the token generation and attest their integrity towards all network devices.

In addition to software attacks, PASTA is also able to detect physical attacks. For this purpose, PASTA relies on the common assumption that physical attacks require an adversary to take a targeted device offline for at least the time period δ_a (e.g., 10 min) [32, 33, 65, 68]. Hence, any prover whose last participation in a token generation is more than or equal to δ_a time ago is considered to be physically compromised. In contrast to SCAPI (Chapter 6), PASTA calculates the absence time of each prover individually, which gives provers twice as much time to participate in the protocol. Thus, compared with SCAPI, PASTA can increase robustness against disruptions while providing the same security level or halve δ_a to provide stronger security guarantees with the same level of robustness.

Physical attacks require not only much time, but also expensive equipment and laborious handwork of highly skilled personnel. In our adversary model (Section 2.4), we therefore argued that a physical adversary may only be able to successfully tamper with a limited number of devices simultaneously. Based on this assumption and the property of tokens to be publicly verifiable, PASTA is able to reunite groups of prover devices that have been separated for longer than δ_a time, meaning that any prover of one group has not generated a token with any prover of the other group within δ_a time and thus violated the original absence assumption. To reunite groups, provers of different groups exchange their generated tokens that (i) testify the permanent presence of all provers of the particular group during the separation, and (ii) could not have been forged by an adversary under the aforementioned assumption due the large number of provers in that group. This way, PASTA can detect physical attacks and recover

from long-lasting network splits that segmented a network in separated groups, as opposed to any other attestation protocol.

Finally, we discuss the security of PASTA and show its scalability in large networks and robustness in dynamic as well as disruptive network topologies.

Outline. The rest of this chapter is organized as follows. In Section 7.2, we explain Schnorr multisignatures, which are a required building block in our protocol. Section 7.3 presents PASTA, a practical attestation protocol for autonomous embedded systems. In Section 7.4, we evaluate the performance and robustness of PASTA. Section 7.5 concludes this chapter.

7.2 SCHNORR MULTISIGNATURES

Our proposed attestation protocol makes use of Schnorr multisignatures to reduce the size of generated signatures and the computational costs to verify them. Schnorr signatures [125] rely on a cyclic group $\mathbb{G}$ of prime order q, a generator g of $\mathbb{G}$, and a hash function $\mathsf{Hash}()$. A signature key pair (x, y) is generated by choosing a random secret key $x \in \mathbb{Z}_q$ and computing the corresponding public key $y = g^x$. In the Schnorr multisignature setting [16], there are n signers, who each possess an individual secret key $x_1, \ldots, x_n$ and a corresponding public key $y_1 = g^{x_1}, \ldots, y_n = g^{x_n}$. In order to collectively sign a common message msg, signers perform the following steps:

1. $r_i \leftarrow \mathsf{GenCommit}()$: $k_i \leftarrow \mathbb{Z}_q$; $r_i = g^{k_i}$.

   Description: Each signer i picks a random secret $k_i \in \mathbb{Z}_q$ and computes a commitment $r_i = g^{k_i}$.

2. $r \leftarrow \mathsf{AggCommit}(r_1; r_2; \ldots; r_n)$: $r = \prod_{i=1}^{n} r_i$.

   Description: The commitment r_i of each signer i is aggregated into a final commitment r by computing $r = \prod_{i=1}^{n} r_i$.

3. $s_i \leftarrow \mathsf{GenSig}(r; msg)$: $c = \mathsf{Hash}(r \| msg)$; $s_i = k_i + cx_i$.

   Description: Each signer i computes the common challenge $c = \mathsf{Hash}(r \| msg)$, which is based on the aggregated commitment r and the message msg to be signed. Afterwards each signer i generates its partial signature $s_i = k_i + cx_i$.

4. $s \leftarrow \mathsf{AggSig}(s_1; s_2; \ldots; s_n)$: $s = \sum_{i=1}^{n} s_i$.

   Description: The signature s_i of each signer i is aggregated in a final signature s by computing $s = \sum_{i=1}^{n} s_i$. The final aggregated signature consists of the tuple (r, s).

To verify the aggregated signature $sig = (r, s)$ with the public keys $y_1, \ldots, y_n$ of all signers, a verifier executes $\mathsf{VerSig}(y_1, \ldots, y_n; sig; msg)$, which comprises the following steps:

1. $\mathsf{y} \leftarrow \mathsf{AggKey}(\mathsf{y}_1, \mathsf{y}_2, ..., \mathsf{y}_n)$: $\mathsf{y} = \prod_{i=1}^{n} \mathsf{y}_i$.

   Description: The public keys of all signers that contributed to (r, s) is aggregated by computing $\mathsf{y} = \prod_{i=1}^{n} \mathsf{y}_i$.

2. *valid / invalid* $\leftarrow \mathsf{Verify}(r, s)$: $c = \mathsf{Hash}(r \| msg)$; $g^s \stackrel{?}{=} r\mathsf{y}^c$.

   Description: The verifier first computes the common challenge c based on r and msg. Finally, the verifier regards the aggregated signature as valid if $g^s = r\mathsf{y}^c$, and in the other case as invalid.

Note that a corrupt signer could set its public key to $\mathsf{y}_1 = g^{x_1}(\prod_{i=2}^{n} \mathsf{y}_i)^{-1}$ and then forge valid signatures on behalf of all n signers. However, this so-called rogue-key attack [16, 104] is irrelevant in our attestation protocol, as all keys are predeployed (Section 7.3.1) and only uncompromised devices are able to participate in the signing process (Section 7.3.2).

7.3 PASTA: PRACTICAL ATTESTATION OF AUTONOMOUS EMBEDDED SYSTEMS

In this section, we describe our attestation protocol PASTA. Compared with our previously proposed attestation protocols, PASTA enables multiple entities to be a verifier and check the integrity of all provers in the network. More specifically, each device $\mathcal{D}_i$ participating in PASTA is a verifier $\mathcal{V}_i$ and can additionally be a prover $\mathcal{P}_i$. This implies that any prover at the same time also acts as a verifier. Provers implement the necessary security requirements (Section 2.3) and perpetually engage in the protocol execution. As opposed to provers, devices that are only verifiers do not need to implement secure hardware and are not required to periodically execute PASTA. Thus, any device, including entities that are potentially untrustworthy or only occasionally connect to the network, e.g., an external network operator, can act as a verifier.

PASTA consists of three different phases. In the *deployment phase*, each device is set up once by the network operator. Afterwards, prover devices in the network continually execute the *token generation phase*, in which provers repeatedly generate tokens. Each token attests the software and hardware integrity of all provers that participated in its generation, at the generation time. Simultaneous to the token generation phase, all devices, i.e., provers and verifiers, perform the *token exchange phase*. In this phase, devices distribute, verify, and validate the generated tokens from the provers. Devices eventually use their verified and validated tokens to determine the integrity of all provers in the network. In the following, we describe the deployment (Section 7.3.1), token generation (Section 7.3.2), and token exchange phase (Section 7.3.3). Finally, we explain the token validation (Section 7.3.4), which is part of the token exchange.

7.3.1 *Deployment Phase*

Initially, devices in the network are deployed and set up by the network operator $\mathcal{O}$. Each device $\mathcal{D}_i$ is equipped with a unique identifier i and the public signature key y_O of $\mathcal{O}$. Devices that are provers additionally store a token signature key pair (x_i, y_i). A prover $\mathcal{P}_i$ uses its private token key x_i to generate tokens that prove its integrity. To verify tokens from $\mathcal{P}_i$, a device $\mathcal{D}_j$ requires the public token key y_i of $\mathcal{P}_i$. Therefore, all devices in the network store the public keys of all prover devices. In addition, any two devices $\mathcal{D}_i$ and $\mathcal{D}_j$ hold a unique symmetric channel key ck_{ij}, which they use to authenticate any communication between them. To save storage, $\mathcal{O}$ may equip devices with a further key pair and a certificate over the public key. This enables devices to establish a channel key on demand and exchange public token keys ad-hoc. Provers store and execute all protocol data and code inside their TEE. Thus, an adversary must perform physical attacks to obtain secrets or tamper with the protocol execution.

Moreover, all devices maintain a token set TS, which initially contains the initialization token T_0. Each token T consists of a timestamp $T.ts$, a list of device identifiers $T.ids$, an aggregated signature $T.sig$, and a Boolean flag $T.valid$. $T.ts$ records the time when the token was generated, $T.ids$ stores the identifiers of all provers that participated in the token generation, $T.sig$ contains the aggregated signature from all participating provers computed over $T.ts$ and $T.ids$, and $T.valid$ indicates whether the device that stores T has successfully validated T ($T.valid = true$) or not. A device $\mathcal{D}_i$ considers T as valid, if $\mathcal{D}_i$ has ensured that all provers listed in $T.ids$ were at time $T.ts$ never offline for longer than the attack time δ_a, hence, were physically healthy at $T.ts$. The validity flag is not protected by the signature $T.sig$ because $T.valid$ is volatile and set by each device itself. Since all provers are assumed to be healthy at protocol start, the initialization token T_0 stores the start time of the protocol in $T_0.ts$, the identifier of all provers in $T_0.ids$, the aggregated signature over $T_0.ts$ and $T_0.ids$ from all provers in $T_0.sig$, and $true$ in $T_0.valid$.

For convenience, we assume the initial deployment of all devices in the network. Nonetheless, $\mathcal{O}$ can enroll a new device $\mathcal{D}_i$ at any later stage. To this end, $\mathcal{D}_i$ is initialized as described, but additionally equipped with a certificate that allows other devices to establish required keys with $\mathcal{D}_i$ on demand. If $\mathcal{D}_i$ is a prover device, it also holds a special token that contains its deployment time and the identifier and signature of $\mathcal{O}$ and $\mathcal{D}_i$. Table 7.2 summarizes relevant definitions.

7.3.2 *Token Generation Phase*

Overview. After deployment, prover devices periodically generate tokens that attest their integrity at the token generation time. To this end, provers recurrently execute the token generation protocol, which consists of two communication rounds.

Acronym	*Usage*
$\mathcal{D}_i/\mathcal{V}_i/\mathcal{P}_i$	device / verifier / prover with identifier i
$\mathcal{O}$	trusted network operator
$\mathsf{y}_{\mathcal{O}}$	public signature verification key of $\mathcal{O}$
x_i / y_i	private / public token key of prover $\mathcal{P}_i$
ck_{ij}	channel key between $\mathcal{D}_i$ and $\mathcal{D}_j$
T	token; T contains: $T.ts, T.ids, T.sig, T.valid$
δ_{gen}	time between periodic token generations
δ_a	time Adv_{hw} must take $\mathcal{D}_i$ offline during attack
β	number of provers Adv_{hw} can compromise in δ_a
$\mathsf{Time}()$	returns current time
$\mathsf{VerifySW}()$	returns if local software state is trustworthy
$\mathsf{IsHealthy}(\mathcal{P}_i)$	returns if $\mathcal{P}_i$ is healthy ($\neq$ compromised)
$\mathsf{ReqNewToken}(\delta)$	returns if token needs to be regenerated

Table 7.2: Notation.

In the first round, a virtual spanning tree is arranged in the network, whose root is a particular initiator prover device that starts the protocol. The tree allows data from the initiator to be efficiently broadcasted to all provers and data from all provers to be efficiently propagated back to the initiator. At first, all provers receive a token generation request from the initiator and then respond with their Schnorr commitments and their device identifiers. Both the identifiers and commitments are collected and aggregated in each hop, and then forwarded to the next hop along the tree towards the initiator. Eventually, the initiator receives the identity of all participating provers and their commitments. In the second round, the initiator broadcasts the aggregated commitments, current time, and prover identifiers, which forms the attestation challenge. Subsequently, provers generate a partial signature over the time and prover identifiers, which is also aggregated and propagated along the tree to the initiator. After collecting all signatures, the initiator assembles the final token T. T contains the time in $T.ts$, the prover identifiers in $T.ids$, and the aggregated signature over $T.ts$ and $T.ids$ in $T.sig$.

During token generation, provers communicate authenticated using their channel key ck, which prevents a network adversary from manipulating the protocol execution. To also protect against compromise of the software and hardware, provers only participate in the token generation if they have ensured their own software integrity and the physical integrity of their neighboring provers in the tree. To verify their own software integrity, provers implement a function $\mathsf{VerifySW}()$, which measures the local state and returns *true* if the

software is in a trustworthy state[2]. We abstract from implementation details of VerifySW() to support a wide range of attestation mechanisms [3, 38, 108, 150]. Recall that any protocol code is executed inside the TEE, such that Adv_{sw} is unable to bypass VerifySW(). Furthermore, provers implement a function IsHealthy($\mathcal{P}_j$), which determines the integrity of provers using tokens that were generated in previous runs of the token generation protocol and were then propagated, verified, and validated in the token exchange phase. IsHealthy($\mathcal{P}_j$) returns *true* if $\mathcal{P}_j$ has proven to be healthy within the last δ_a time and *false* otherwise. Since Adv_{hw} requires at least δ_a time to perform physical attacks, provers that pass IsHealthy($\mathcal{P}_j$) are physically healthy at the present time. Details of IsHealthy($\mathcal{P}_j$) are given in the next subsection (Section 7.3.3).

In the following, we explain both rounds of the token generation protocol in detail.

Round 1: Initialization. To ensure that prover devices periodically engage in the token generation, each prover monitors the freshness of its own tokens. A prover device initiates a token generation, whenever it notices that the newest token T in whose generation it participated is older than a specific generation interval δ_{gen}, as indicated by the function ReqNewToken(δ_{gen}), shown in Algorithm 7.1.

Algorithm 7.1: $\mathcal{P}_i$ determines if it must initiate a token generation.

```
1: procedure ReqNewToken(δ)
2:     for T ∈ TS do
3:         if i ∈ T.ids and T.ts > Time() − δ then
4:             return false
5:     return true
```

The generation interval δ_{gen} must be a fraction of δ_a ($\delta_{gen} < \delta_a$) to ensure that each prover participates at least once in a token generation within δ_a time. A prover that fulfills this requirement has never been offline for longer than δ_a time and is therefore considered to be physically healthy by all network devices. A small δ_{gen} enhances resilience to network disruptions, as tokens are generated more frequently, but on the downside increases communication and computational effort.

As illustrated in Figure 7.1, the token generation protocol starts with a prover $\mathcal{P}_i$ that checks whether it requires a new token and is in a trustworthy software state, using ReqNewToken(δ_{gen}) and VerifySW(). If both checks pass, $\mathcal{P}_i$ will initiate a token generation in the network. To this end, $\mathcal{P}_i$ stores its identifier i in ids_i and the current time in ts, which then both form the token initiation message msg_{init}. Furthermore, $\mathcal{P}_i$ generates its Schnorr commitment r_i over a random secret (Section 7.2). Next, any neighboring prover $\mathcal{P}_j$ that is in a physically

2 Depending on the implementation, the software state may actually be measured prior to the invocation of VerifySW(). For instance, a TPM computes hash values over software binaries before they are loaded [108], so that VerifySW() only compares already performed measurements from the TPM with known good reference values.

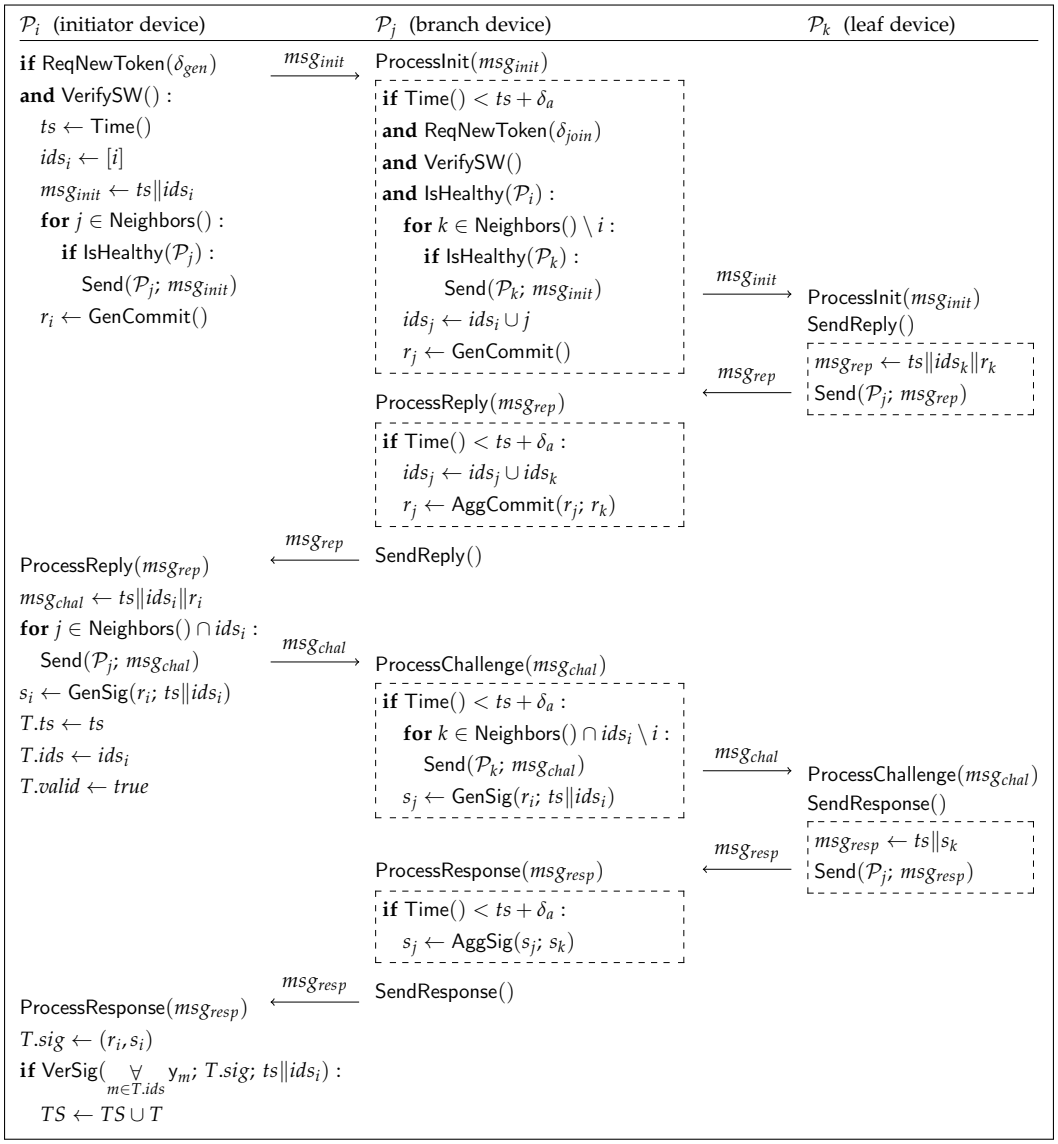


Figure 7.1: Illustration of the token generation protocol. Prover $\mathcal{P}_i$ initiates the token generation, whereupon a spanning tree is constructed. $\mathcal{P}_j$ represents a branch device, which is connected to $\mathcal{P}_i$ and $\mathcal{P}_k$. $\mathcal{P}_k$ is a leaf device, as it is only connected to $\mathcal{P}_j$. When receiving messages, provers unpack their content, e.g., ts and ids_i from msg_{init}. For clarity, we omitted the authentication of messages. In practice, all exchanged messages between any two provers $\mathcal{P}_i$ and $\mathcal{P}_j$ are authenticated by the sender and verified by the receiver using the channel key ck_{ij}.

healthy state, determined by $\mathcal{P}_i$ using $\mathsf{IsHealthy}(\mathcal{P}_j)$, is sent msg_{init}. All messages are authenticated by the sender and verified by the receiver using the respective channel key. Invalid messages are dropped and not processed. For clarity, we omitted message authentication in Figure 7.1.

A prover device $\mathcal{P}_j$ that receives a message msg_{init} initially ensures that the token generation was initiated within the last δ_a time, as $\mathsf{Time}() < ts + \delta_a$. This time check is executed whenever provers receive a message in the token generation protocol to prevent a network adversary from replaying recorded messages from previous runs of the protocol. Next, $\mathcal{P}_j$ checks whether it would soon require a token generation, indicated by a call to the function $\mathsf{ReqNewToken}()$ with a specific join interval δ_{join} that is smaller than the generation interval δ_{gen} ($\delta_{join} < \delta_{gen}$). Using a smaller interval to join a token generation than to initiate one saves overhead, as it prevents the costly initiation of multiple token generations in close succession. Afterwards, $\mathcal{P}_j$ checks whether the prover from which it received msg_{init} is physically healthy and itself is in a trustworthy software state. If both checks also pass, $\mathcal{P}_j$ joins the token generation session. To this end, $\mathcal{P}_j$ stores its own identifier j and the received identifiers ids_i in its own ids_j and generates its Schnorr commitment r_j. Additionally, $\mathcal{P}_j$ invites further healthy neighboring provers to join the current token generation session by propagating msg_{init}. Note that provers only ensure the integrity of their neighboring provers, but not of all provers that participate in the token generation. This is because provers can rely on the trustworthiness of healthy neighbors and be confident that they in turn ensure the physical integrity of their neighbors. Thus, a chain of trust is established that prevents any physically compromised prover from participating in the token generation.

Prover devices that receive a msg_{init} from $\mathcal{P}_j$ likewise perform the same steps as $\mathcal{P}_j$. This way, beginning with the initiator device $\mathcal{P}_i$, a spanning tree is arranged in the network, where parent provers invite their children to join the token generation. Eventually, msg_{init} is received by leaf devices that have no children because all their neighbors are already participating in the token generation are compromised or recently generated a token. Receiving msg_{init}, each leaf device $\mathcal{P}_k$ answers its parent with a msg_{rep} message, which contains the token generation timestamp ts, the prover identifier ids_k, and the Schnorr commitment r_k. A prover $\mathcal{P}_j$ that receives a msg_{rep} from a child prover $\mathcal{P}_k$ merges the received device identifiers ids_k with its stored identifiers in ids_j and aggregates the received Schnorr commitment r_k with its stored commitment in r_j (Section 7.2). After processing the msg_{rep} from all child provers, provers create their own msg_{rep} and send it to their own parent devices, which in turn perform the same steps. Finally, the initiator prover $\mathcal{P}_i$ receives and aggregates the identifiers and Schnorr commitments of all provers that are participating in the token generation session.

Round 2: Finalization. After the first round, the only data that the initiator prover $\mathcal{P}_i$ is missing to generate the final token T, is the aggregated signature s of all participating provers, computed over $T.ts$ and $T.ids$. To collect s, $\mathcal{P}_i$ prepares a message msg_{chal}, which stores ts, ids_i, and r_i. Next, msg_{chal} is propagated in the

network from device to device, using the tree topology from the first round. With the content of msg_{chal}, each participating prover $\mathcal{P}_j$ then computes its partial signature s_j (Section 7.2). Afterwards, partial signatures are forwarded along the tree topology back to the initiator $\mathcal{P}_i$ and are aggregated in each hop, like commitments in the first round. $\mathcal{P}_i$ eventually receives and aggregates the partial signatures of all participating provers in s_i. Finally, $\mathcal{P}_i$ builds the token T, which stores ts in $T.ts$, ids_i in $T.ids$, *true* in $T.valid$, and the tuple (r_i, s_i) in $T.sig$. In case $T.sig$ is valid, $\mathcal{P}_i$ adds T to its token set TS. Thus, T attests that all provers listed in $T.ids$ have been healthy at $T.ts$. Finally, $\mathcal{P}_i$ uses the token exchange phase (Section 7.3.3) to distribute T in the network.

Remarks. For clarity, we described the token generation in a simplified version. In practice, timers and error messages must prevent the protocol from hanging if messages are lost or replayed, verification checks fail, or provers are invited to join the same token generation multiple times. Furthermore, to support network dynamics during token generation, an ad-hoc routing protocol must enable each device to reach its (initial) neighboring devices in the virtual tree topology. Additionally, we recommend that provers store different values for δ_{gen}. For instance, a dedicated leader prover may use a lower δ_{gen} than other provers. This prevents multiple provers from initiating a token generation simultaneously, which would result in the unnecessary generation of multiple tokens. Instead, the dedicated leader prover would always initiate the token generation and other provers would only take over, if the leader is unavailable.

7.3.3 *Token Exchange Phase*

Token Exchange. After executing the token generation protocol, the initiator device holds a new token that testifies the integrity of one or multiple provers at the token generation time. To make this information available to all network devices, generated tokens are propagated and stored by all devices. For this purpose, devices within direct communication range continuously synchronize their token set TS. A synchronization is initiated when devices connect to each other or a connected neighboring device just generated or received a new token. For efficiency, devices only exchange tokens that the receiver is missing. To determine the missing tokens, devices initially compare the number and checksums of their stored tokens.

Any device, including untrustworthy devices, can participate in the exchange of tokens and act as a data mule. This is feasible and secure, as the integrity and authenticity of each token T is protected by its contained aggregated signature $T.sig$. Devices verify the signatures of all received tokens. Corrupt tokens, which fail the verification, are discarded and are not added to the token set TS. Note that the signature verification only protects against Adv_{sw}, but not Adv_{hw}, who is able to forge token signatures of provers after physically tampering with them. Attacks from Adv_{hw} are prevented by the token validity flags, which are not

transferred during token exchange. Instead, each device sets $T.valid$ to *false* for any token T that is received and added to TS. This indicates that the device storing T has not (yet) ensured that the provers listed in $T.ids$ have never been offline for longer than δ_a time. Because provers listed in invalid tokens may potentially be in a compromised state, invalid tokens are disregarded when determining the integrity of provers with $\mathsf{IsHealthy}()$. To determine and set the validity of stored tokens, each device performs a so-called token validation after it has synchronized all tokens with its neighboring devices, which is described in detail in the next subsection (Section 7.3.4). The token validation checks whether Adv_{hw} has been unable to physically tamper with all provers listed in $T.ids$ based on the limitations of Adv_{hw} (Section 2.4), the timestamp in $T.ts$, the current time, and the chain of trust derived from already validated stored tokens. Any token T that passes these checks is marked as valid, with $T.valid$ set to *true*. Tokens that could not be validated are still kept, i.e., not discarded from TS, since tokens received in the future may prove their validity.

In case a receiver $\mathcal{D}_i$ of tokens determines with $\mathsf{IsHealthy}(\mathcal{P}_j)$ that the sender $\mathcal{P}_j$ is a healthy prover, $\mathcal{D}_i$ can take advantage of $\mathcal{P}_j$'s trustworthiness. To this end, transmitted tokens are protected by a MAC using the channel key ck_{ij} of involved devices $\mathcal{D}_i$ and $\mathcal{P}_j$. Thus, $\mathcal{D}_i$ can rely on the authenticity of the channel as well as the correct behavior of $\mathcal{P}_j$. This enables $\mathcal{D}_i$ to omit verifying the signature of received tokens as well as transferring the token validity flags from $\mathcal{P}_j$, which both saves computational resources.

Devices routinely discard tokens whose timestamp is so old that they neither play a role in determining the healthiness of provers ($T.ts \leq \mathsf{Time}() - \delta_a$) nor in establishing validity in other tokens ($T.ts \leq \mathsf{Time}() - p/\beta \cdot \delta_a$), with β being the concurrency factor and p the total number of provers.

Determining the Integrity of Provers. Devices that have synchronized and validated their token set TS can afterwards determine the integrity of provers. To this end, devices execute the function $\mathsf{IsHealthy}(\mathcal{P}_i)$, which is shown in Algorithm 7.2.

Algorithm 7.2: A device determines the integrity of a prover $\mathcal{P}_i$.

```
1: procedure IsHealthy(P_i)
2:     for T ∈ TS do
3:         if i ∈ T.ids and Time() < T.ts + δ_a and T.valid then
4:             return true
5:     return false
```

Devices that execute $\mathsf{IsHealthy}(\mathcal{P}_i)$ check whether $\mathcal{P}_i$ participated in the generation of a token T whose validity has been ensured and which was generated within the last δ_a time. A prover $\mathcal{P}_i$ that passes these checks has proven to be in a trustworthy software state between the time $T.ts$ and now. In addition, $\mathcal{P}_i$ is physically healthy at the present time, as T testifies the physical integrity of $\mathcal{P}_i$ at $T.ts$ and successful physical attacks require at least the attack time δ_a.

According to the adversary model (Section 2.4), our attestation protocol is secure, if Adv_{sw} and Adv_{hw} are unable to fake a healthy system state for a prover $\mathcal{P}_a$ that is at the time of its own attestation in a compromised state. To fake a healthy state for $\mathcal{P}_a$, Adv_{hw}, hence, also the weaker Adv_{sw}, would need trick a device $\mathcal{D}_i$ into storing a token T that passes $\mathsf{IsHealthy}(\mathcal{P}_a)$. To this end, Adv_{hw} must forge T in a way that it contains a valid signature for $\mathcal{P}_a$, since $\mathcal{D}_i$ will only accept T during token exchange, if T passes the signature verification. To forge a valid token signature for $\mathcal{P}_a$, Adv_{hw} must possess the private token key x_a of $\mathcal{P}_a$. By performing network attacks or compromising the software on $\mathcal{P}_a$, Adv_{hw} is unable to get access to x_a. This is because all secrets and protocol code are stored and execute inside the TEE of provers and never leave the TEE. In addition, the TEE is immutable by compromised software (Section 2.3) and the token generation protocol code enforces that provers with a compromised software quit the protocol execution. Hence, a software-compromised prover $\mathcal{P}_a$ is unable to access its private token key x_a. However, Adv_{hw} may also physically compromise $\mathcal{P}_a$ to gain access to x_a, which enables Adv_{hw} to forge valid token signatures on behalf of $\mathcal{P}_a$. Yet, because Adv_{hw} must take $\mathcal{P}_a$ offline for δ_a time during the physical attack, $\mathcal{P}_a$ will not generate a token for more than δ_a time. Thus, after the physical attack is completed, all devices will only store outdated tokens from $\mathcal{P}_a$ and regard $\mathcal{P}_a$ as compromised when executing $\mathsf{IsHealthy}(\mathcal{P}_a)$. As a result, $\mathcal{P}_a$ is from then on neither able to participate in the token generation with other healthy provers, nor able to issue a token that will pass the token validation on healthy devices, as described in the following subsection (Section 7.3.4).

Note that devices cannot distinguish between an unreachable and a compromised prover. We assume that Adv_{hw} has full control over the network and, thus, can prevent the generation and exchange of tokens in the network. Hence, if there is no token T that attests the integrity of a prover $\mathcal{P}_i$, a device must assume that $\mathcal{P}_i$ is in a compromised state.

7.3.4 *Token Validation*

Overview. The token validation is the essential part of PASTA to detect physical attacks. Recall that a stored validated token T testifies that all provers listed in $T.ids$ were physically healthy at $T.ts$. By contrast, the authenticity of stored invalid tokens has not been ensured (yet), meaning that each invalid token could have been forged by Adv_{hw} and list physically compromised provers in $T.ids$ and a bogus timestamp in $T.ts$. During token validation, a device $\mathcal{D}_i$ updates the validity flags of all stored invalid tokens, whereby any invalid token for which $\mathcal{D}_i$ can rule out that it is forged by Adv_{hw} is marked as valid. More specifically, $\mathcal{D}_i$ uses its already validated tokens to build a chain of trust between all physically healthy provers, which are provers that were never offline for longer than the attack time δ_a. Starting with the initialization token pre-deployed as valid, $\mathcal{D}_i$ iteratively uses the information from already validated tokens to validate further invalid tokens. In this process, $\mathcal{D}_i$ makes use of the time and simultaneously

Algorithm 7.3: $\mathcal{D}_i$ validates its stored tokens in step one.

```
1: procedure ValidateTime()
2:     hdevs ← ∅                                   ▷ provers considered healthy
3:     for i = 1, 2, ..., n do
4:         if IsHealthy(P_i) then
5:             hdevs ← hdevs ∪ i
6:     for T ∈ TS do
7:         if (T.valid = false) and (T.ids ∩ hdevs ≠ ∅) then
8:             T.valid ← true
9:             go to 3
```

limitations of Adv_{hw}, which are stated in (1) and (2) in our adversary model (Section 2.4). In case $\mathcal{D}_i$ attempts to validate a forged token T_a, $\mathcal{D}_i$ will be unable to establish a chain of trust for provers listed in T_a, since they have been offline for at least δ_a time during the physical attack. Consequently, $\mathcal{D}_i$ will not mark T_a as valid.

The token validation consists of two steps. In each step, tokens are validated taking assumption (1), respectively (2), on Adv_{hw} into account. Below, we describe both steps in detail.

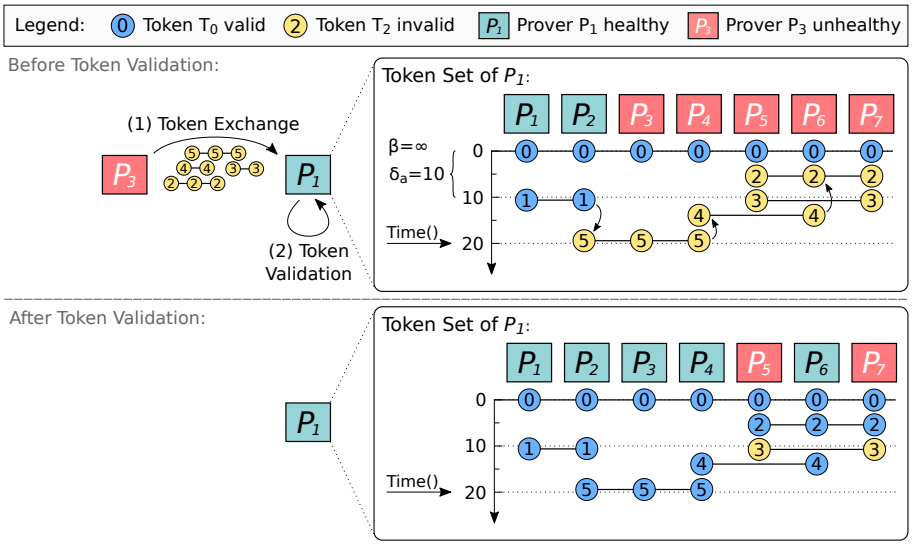
(1) Validation of Time Assumption. In the first step, already validated tokens are used to determine the set of healthy provers. With the set of healthy provers, invalid tokens are then validated. The process is illustrated in Algorithm 7.3 and explained in the following.

2-5: Initially, a device $\mathcal{D}_i$ determines and stores the identifiers of all provers that $\mathcal{D}_i$ considers to be healthy in *hdevs*. For this purpose, $\mathcal{D}_i$ uses the function IsHealthy(), which we described in the previous subsection (Section 7.3.3).

6-9: Next, $\mathcal{D}_i$ makes use of *hdevs* to validate tokens from its token set TS. In detail, the validity flag $T.valid$ of any token T that lists a healthy prover, which is the case if $T.ids \cap hdevs \neq \emptyset$, is set to *true*. This is because healthy provers only engage in the token generation with other healthy provers. Since *hdevs* testifies that at least one prover in $T.ids$ is healthy, all provers in $T.ids$ must be healthy. Thus, T cannot have been forged by Adv_{hw} and is set valid. Next, $\mathcal{D}_i$ starts the procedure all over again. This is necessary, as T may now testify the integrity of additional provers that are currently not contained in *hdevs*. With the extended *hdevs*, already processed tokens that were not validated in the current iteration of the procedure may be validated in the next iteration.

For clarity, Figure 7.2 shows an example for the validation of the time assumption.

(2) Validation of Simultaneity Assumption. In the second step, devices additionally make use of the assumption that Adv_{hw} is unable to compromise more



Prover $\mathcal{P}_1$ receives the tokens T_2, T_3, T_4, and T_5 from prover $\mathcal{P}_3$ at time 20. Before the token validation, $\mathcal{P}_1$ considers $\mathcal{P}_3$, $\mathcal{P}_4$, $\mathcal{P}_5$, $\mathcal{P}_6$, and $\mathcal{P}_7$ to be compromised, since $\mathcal{P}_1$ does not store a validated and recent (i.e., less than $\mathsf{Time}() - \delta_a$ time old) token that lists $\mathcal{P}_3$, $\mathcal{P}_4$, $\mathcal{P}_5$, $\mathcal{P}_6$, or $\mathcal{P}_7$. During token validation, $\mathcal{P}_1$ iteratively establishes a chain of trust based on already validated tokens, as indicated by the arrows. Initially, token T_1 testifies the integrity of $\mathcal{P}_1$ and $\mathcal{P}_2$ at time $T_1.ts \approx 11$. Because Adv_{hw} would have had too little time to tamper with $\mathcal{P}_1$ or $\mathcal{P}_2$ between $T_1.ts$ and now, both provers must be physically healthy at the present moment. Since $\mathcal{P}_2$ must be healthy and is listed in T_5, T_5 cannot be forged by Adv_{hw}, hence, is set valid. Next, the newly validated T_5 testifies the integrity of $\mathcal{P}_4$, which is why token T_4 is set valid. Finally, T_4 testifies that $\mathcal{P}_6$ is healthy, so that T_2 is set valid. Token T_3 cannot be validated, since $\mathcal{P}_1$ does not store a validated and recent token that testifies the integrity of provers listed in T_3, i.e., $\mathcal{P}_5$ and $\mathcal{P}_7$, at the present time. Thus, T_3 may be forged by Adv_{hw}, hence, remains invalid. After token validation, $\mathcal{P}_1$ regards $\mathcal{P}_2$, $\mathcal{P}_3$, $\mathcal{P}_4$, and $\mathcal{P}_6$ as healthy, due to the newly received and validated tokens.

Figure 7.2: Illustration of the token validation from the perspective of a prover $\mathcal{P}_1$. This scenario represents a highly disrupted network, where provers only rarely had a connection to each other. Thus, provers were unable to periodically perform the token generation as a group. The concurrency factor β is set to ∞, which means that Adv_{hw} can physically compromise all provers concurrently.

than β provers per δ_a time concurrently. Based on this assumption, validated tokens that are outdated, i.e., older than δ_a, can be used to validate remaining invalid tokens from the first token validation step. This is possible, as outdated validated tokens may testify the healthiness of so many provers at a past time that Adv_{hw} would be unable to have physically compromised all of them at the present time. Therefore, newer (yet) invalid tokens generated by one of those healthy provers must be valid.

More specifically, each validated token T_v testifies the healthiness of all provers listed in T_v at time $T_v.ts$. At the present time, Adv_{hw} can at the maximum have physically compromised $maxcdevs = \lfloor(\mathsf{Time}() - T_v.ts)/\delta_a\rfloor \cdot \beta$ provers of T_v. If an invalid token T_i shares more than $maxcdevs$ provers with T_v, i.e., $|T_i.ids \cap T_v.ids| > maxcdevs$, then T_i must be valid. This is because to forge T_i, Adv_{hw} has to compromise all provers listed in T_i, which is, however, a contradiction

Algorithm 7.4: $\mathcal{D}_i$ validates its stored tokens in step two.

```
1:  procedure ValidateSimultaneity()
2:      for T_i ∈ TS do
3:          if T_i.valid = false then                          ▷ attempt to validate T_i
4:              itoks ← {i}
5:              idevs ← T_i.ids
6:              for T_k ∈ TS do
7:                  if (k ∉ itoks) and (T_k.ts > T_i.ts) then
8:                      maxcdevs ← ⌊(Time() − T_i.ts)/δ_a⌋ · β
9:                      if |T_k.ids ∩ idevs| > maxcdevs then
10:                         itoks ← itoks ∪ k
11:                         idevs ← idevs ∪ T_k.ids
12:                         go to 6
13:             vdevs ← ∅
14:             for T_v ∈ reverse(sort(TS)) do                 ▷ highest T.ts first
15:                 if (T_v.valid) and (T_v.ids ∩ idevs ≠ ∅) then
16:                     maxcdevs ← ⌊(Time() − T_v.ts)/δ_a⌋ · β
17:                     vdevs ← vdevs ∪ (T_v.ids ∩ idevs)
18:                     if |vdevs| > maxcdevs then             ▷ itoks valid
19:                         for k ∈ itoks do
20:                             TS.T_k.valid = true
21:                         ValidateTime()
22:                         go to 2
```

to the fact that at least one prover in $T_i.ids \cap T_v.ids$ must be healthy. A larger intersection $|T_i.ids \cap T_v.ids|$ enables T_v to be older and still be used for validation, which increases robustness against device and network disruptions. To maximize this intersection, T_i and T_v can be extended by further tokens from TS that share the same device identifiers with either T_i or T_v. This way, two device identifier sets, *idevs* and *vdevs*, of invalid (*idevs*) and valid (*vdevs*) tokens can be used, which results in the extended intersection $idevs \cap vdevs$. The process is illustrated in Algorithm 7.4 and explained in the following.

2-5: A device $\mathcal{D}_i$ iteratively selects an invalid token T_i from TS and attempts to validate T_i in the following steps. Initially, the index i of T_i is recorded in *itoks* and provers listed in T_i are stored in *idevs*.

6-12: Next, *itoks* is iteratively extended by further tokens to be validated, and *idevs* is enlarged by their listed prover identifiers. In this process, *itoks* is constructed in a way that a single valid token in *itoks* is capable of testifying the validity of all tokens in *itoks*. To this end, TS is examined for tokens that (i) are not contained in *itoks* and are more recent than $T_i.ts$, and (ii) share more provers with *idevs* than Adv_{hw} can compromise between $T_i.ts$ and now. A token T_k fulfills (i) if $k \notin itoks$ and $T_k.ts > T_i.ts$. To fulfill (ii), $|T_k.ids \cap idevs|$ must be larger than $maxcdevs = \lfloor(\mathsf{Time}() - T_i.ts)/\delta_a\rfloor \cdot \beta$. If

both checks pass, *itoks* is extended by k and *idevs* by $T_k.ids$. Afterwards, this step (6-12) is repeated.

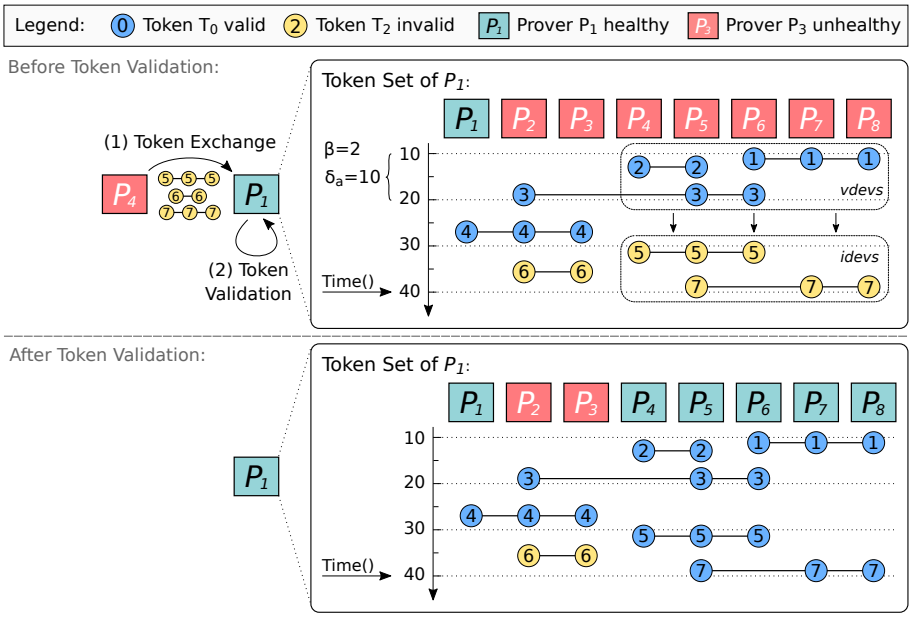
The checks (i) and (ii) ensure that if all tokens in *itoks* are valid, T_k must be valid, and vice versa. This is because between now and the time $T_i.ts$, being according to (i) the oldest of all tokens *itoks* and T_k, Adv_{hw} can only compromise *maxcdevs* provers. Since more than *maxcdevs* provers are according to (ii) contained in $idevs \cap T_k.ids$, Adv_{hw} would have had too little time to tamper with all provers $idevs \cap T_k.ids$, which Adv_{hw} requires to forge T_k or all *itoks* tokens. By ensuring that each token T_k added to *itoks* fulfills (i) and (ii), *itoks* obtains the property that a single valid token in *itoks* is able to testify the validity of all tokens in *itoks*. This property is important for the following step (13-22).

13-22: Next, $\mathcal{D}_i$ determines which provers $vdevs \subseteq idevs$ were listed in already validated tokens, hence, were healthy at a past time t. If the amount of *vdevs* is greater than $maxcdevs = \lfloor(\mathsf{Time}() - t)/\delta_a\rfloor \cdot \beta$, at least one prover in *idevs* must be healthy at the current time. Thus, at least one token from *itoks*, namely the token listing the healthy prover, must be valid. This implies that all tokens in *itoks* must be valid due to the property of *itoks*.

In detail, validated tokens in TS are examined in descending order regarding their timestamp. For a token T_v that attests the integrity of provers in *idevs* at an earlier time $T_v.ts$, *maxcdevs* amounts to $\lfloor(\mathsf{Time}() - T_v.ts)/\delta_a\rfloor \cdot \beta$. For the first found token T_v that attests the integrity of provers in *idevs*, *vdevs* amounts to the intersection between *idevs* and $T_v.ids$: $vdevs = idevs \cap T_v.ids$. However, because provers that are healthy at $T_v.ts$ are also healthy at $T_{v'}.ts < T_v.ts$, *vdevs* inherits all provers from previous iterations for any subsequently found token $T_{v'}$ that attests the integrity of provers in *idevs*: $vdevs = vdevs \cup (T_{v'}.ids \cap idevs)$. If it is eventually ensured that *idevs* are healthy, as $|vdevs| > maxcdevs$, all tokens from *itoks* are set valid. Since the newly validated tokens may be able to change the validity of stored invalid tokens, both steps of the token validation are executed again.

For clarity, Figure 7.3 shows an example for the validation of the simultaneity assumption.

Further Solutions to Validate Tokens. The described token validation works well for devices that regularly exchange and validate tokens. However, some devices may only occasionally connect to the network to check the integrity of provers, such as, for instance, a verifier device from the network operator. Depending on their absence time and the parameters δ_a and β, such devices may have issues establishing the chain of trust in healthy provers and, therefore, may falsely regard healthy provers as compromised. In fact, for a proper token validation, each device needs to reconnect to the network at least every $\delta_{ex} < \lceil p/\beta\rceil \cdot \delta_a$ time, with p being the total number of healthy provers in the network. For instance, in a network with 1000 provers, an attack time δ_a of 10 min, and an



Prover $\mathcal{P}_1$ receives the tokens T_5, T_6, and T_7 from prover $\mathcal{P}_4$ at time 40. The concurrency factor β is 2, which enables $\mathcal{P}_1$ to validate tokens based on the simultaneity assumption. As indicated by the arrows, $\mathcal{P}_1$ uses the outdated ($T.ts < \mathsf{Time}() - \delta_a = 30$) but already validated tokens T_1, T_2, and T_3 to validate the newly received tokens T_5 and T_7. In detail, T_1, T_2, and T_3 testify that $vdevs = \{\mathcal{P}_4, \mathcal{P}_5, \mathcal{P}_6, \mathcal{P}_7, \mathcal{P}_8\}$ were physically healthy at (at least) $T_1.ts \approx 11$, which is the oldest timestamp in T_1, T_2, and T_3. Between $T_1.ts$ and now, Adv_{hw} could at maximum have physically compromised $maxcdevs = \lfloor(\mathsf{Time}() - T_1.ts)/\delta_a\rfloor \cdot \beta = 4$ provers of $vdevs$. Because $vdevs$ contains 5 provers, at least one prover of $vdevs$ must be healthy at the present time. This directly implies that T_5 or T_7 (or both) must be valid. For instance, assuming that $\mathcal{P}_4$ is healthy, T_5 must be valid. However, since T_5 and T_7 both list $\mathcal{P}_5$, the validity of T_5 further implies the validity of T_7 and vice versa. This is because assuming that either T_5 or T_7 is valid, Adv_{hw} would have too little time to physically tamper with $\mathcal{P}_5$ after $T_5 \approx 31$ or $T_7 \approx 39$, which is among others necessary to forge T_5 or T_7. Thus, both T_5 and T_7 must be genuine and are marked as valid. By contrast, $\mathcal{P}_1$ is unable to validate token T_6, generated by $\mathcal{P}_2$ and $\mathcal{P}_3$. This is because between $T_4.ts \approx 27$, at which $\mathcal{P}_2$ and $\mathcal{P}_3$ were healthy, and now, Adv_{hw} would have had enough time to physically compromise $\mathcal{P}_2$ and $\mathcal{P}_3$ and forge token T_6.

Figure 7.3: Illustration of the token validation from the perspective of $\mathcal{P}_1$ and with $\beta = 2$.

attack concurrency factor β of 5, δ_{ex} amounts to 33.3 hours. In case a device is absent from the network for longer than δ_{ex} time, we propose two alternative solutions to validate tokens.

The first solution is to fall back to the approach of existing attestation protocols that detect physical attacks like DARPA [68] or SCAPI (Section 6). They build on assumption (3) of our adversary model (Section 2.4), which states that Adv_{hw} is unable to physically compromise more than a specific amount λ of provers in the network. Based on this assumption, a secure attestation of $p - \lambda + 1$ provers can be guaranteed. To this end, a device $\mathcal{D}_i$ determines all provers $pdevs$ that are listed in tokens with a timestamp more recent than δ_a: $T.ts > \mathsf{Time}() - \delta_a$. If $|pdevs|$ is greater than $p - \lambda$, all $pdevs$ provers must be healthy. Subsequently, $\mathcal{D}_i$ marks all stored tokens that list a prover from $pdevs$ as valid.

The second solution is applicable in case provers provide some form of physical tamper evidence. In this case, the integrity of *hdevs* provers is ensured by physically inspecting the particular provers. Next, all tokens that list a prover from *hdevs* are set valid. Finally, all other stored tokens are validated by executing our proposed token validation.

7.4 EVALUATION

In this section, we first describe our implementation and show measurements conducted on low-end embedded devices (Section 7.4.1). Next, we evaluate the scalability of PASTA by presenting simulations of static networks (Section 7.4.2). Finally, we provide simulation results of PASTA in highly dynamic and disruptive networks to assess its robustness (Section 7.4.3).

7.4.1 *Implementation and Measurements*

Implementation. As a target platform for our implementation, we employed four ESP32-PICO-KIT V4 development boards. The core of the boards constitute a 7x7x0.94mm^2 ESP32-PICO-D4 system-in-package module, which features Wi-Fi and Bluetooth functionalities, 4 MB flash memory, and a 240 MHz dual-core 32-bit microprocessor. Due to its ultra-small size and low-energy consumption, the ESP32-PICO-D4 is well suited for space-limited or battery-operated applications, e.g., as wearable, medical, or sensor devices. Furthermore, it provides the Secure Boot feature and thus the minimal hardware properties required for remote attestation [44, 127].

We used SHA-256 as a cryptographic hash function and a Keyed-Hash Message Authentication Code (HMAC) based on SHA-256 for message authentication. To implement both, we made use of the mbed TLS code [1]. Our implementation of the Schnorr multisignature scheme is based on the Bitcoin cryptographic code [56], which offers a library for elliptic curve operations on curve secp256k1.

Measurements. We found out that the runtime of PASTA is dominated by the cryptographic algorithms and network performance. Table 7.3 depicts runtime measurements of the employed cryptographic algorithms. To attest its integrity, a prover device hashes its firmware and generates a token using the Schnorr multisignature scheme, which consumes around 40.1 ms of runtime in total with a firmware size of 50 kB. The runtime required to verify the integrity of provers listed in a received token depends on various factors. If the token is obtained from an untrustworthy device and the receiver does not store the aggregated public key of all m provers listed in the token, hence, must compute the key ad-hoc, the computation consumes $20.95 + 0.11m$ ms. If the receiver uses a stored aggregated key, the runtime amounts to 20.95 ms. If the token is received from a healthy (i.e., trustworthy) prover, verifying the integrity requires less than 0.06 ms, as only the authenticity of the received message, but not the token

Algorithm	Function	Runtime
Schnorr MuSign	$\mathsf{GenCommit}(k \in \mathbb{Z}_q;\ r = g^k)$	21.284 ms
	$\mathsf{AggCommit}(r = r_1 \cdot r_2)$	0.109 ms
	$\mathsf{GenChallenge}(c = \mathsf{Hash}(r \Vert msg))$	3.819 ms
	$\mathsf{GenSig}(s = k + c \cdot \mathsf{x})$	2.449 ms
	$\mathsf{AggSig}(s = s_1 + s_2)$	0.006 ms
Schnorr MuVerify	$\mathsf{AggKey}(\mathsf{y} = \mathsf{y}_1 \cdot \mathsf{y}_2)$	0.109 ms
	$\mathsf{Verify}(c = \mathsf{Hash}(r \Vert msg);\ g^s \stackrel{?}{=} r\mathsf{y}^c)$	20.896 ms
HMAC-SHA-256	$\mathsf{HMAC}(16\,\mathrm{B})$	0.042 ms
	$\mathsf{HMAC}(1024\,\mathrm{B})$	0.301 ms
SHA-256	$\mathsf{Hash}(51\,200\,\mathrm{B})$	13.171 ms

Table 7.3: Cryptographic runtime measurements on the ESP32.

signature, needs to be verified. For communication between devices, we used the Wi-Fi communication capabilities of the ESP32-PICO-D4. Unfortunately, our measurements were significantly below the stated theoretical throughput of 150 MBit/s, as we measured an average throughput of 12.51 MBit/s on the application layer using TCP and an average round trip time of 4.63 ms.

Memory Consumption. Each device stores its id (4 B), signature key (32 B), public key of $\mathcal{O}$ (64 B), channel key of each prover (16m B), and public key of each prover (64m B). Each stored token consumes between 69 B and $69 + m/8$ B (worst-case). Assuming a network with 10000 provers, our scheme consumes at most 781.4 kB + $|token|$·1.28 kB. The memory consumption can be reduced by establishing channel keys and public keys on demand, and storing aggregated public keys.

Conclusion. We showed that PASTA imposes a low computational complexity and memory consumption on each device. This makes PASTA practical, even on low-end embedded devices. Compared with SANA [7], the only protocol that allows a scalable attestation of many provers towards untrustworthy verifier devices (but is centralized and only detects software attacks), PASTA requires at least one order of magnitude less computational overhead to generate the attestation result and two orders of magnitude less to verify it.

7.4.2 *Static Network Simulations*

Simulation Setup. To evaluate PASTA in large networks, we performed network simulations with the OMNeT++ [146] event simulator. We implemented PASTA on the application layer and used delays based on our runtime and network measurements. On lower network layers, we applied a simplified communication

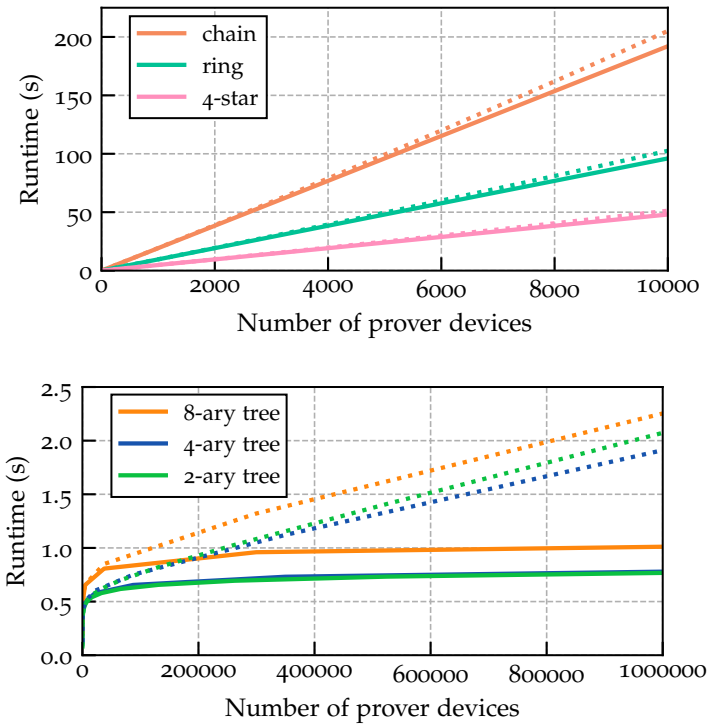
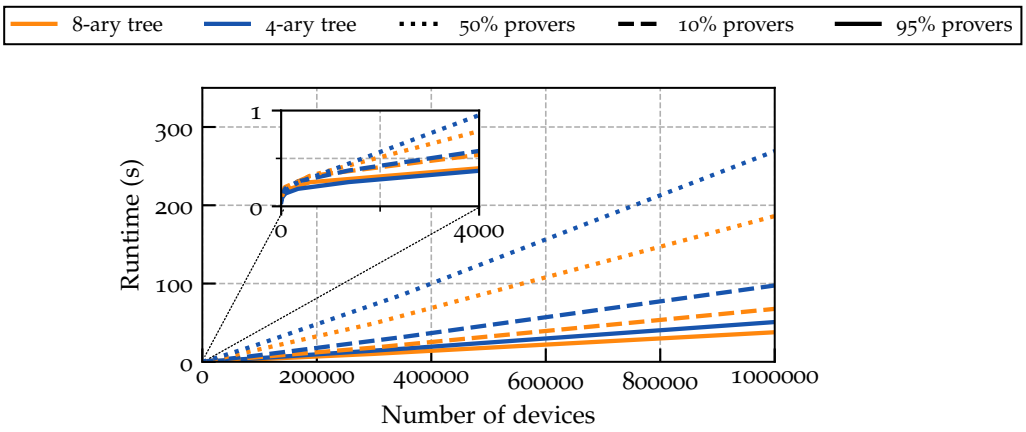


Figure 7.4: Runtime of a token generation in various static topologies. Dotted lines represent topology changes between iterations of the token generation.

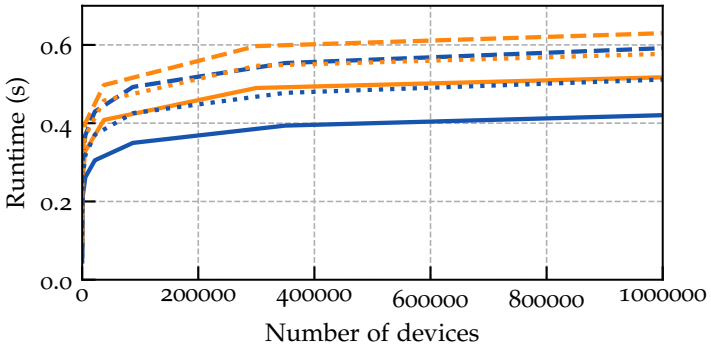
model that enables devices within direct communication range unimpaired half-duplex communication, as long as no other device within range transmits data.

Token Generation. To examine the scalability of PASTA, we conducted simulations in static networks. In these networks, each device is connected to the overall network topology and connections between devices remain (almost) fixed. Figure 7.4 shows our simulation results for the runtime of the token generation phase with a varying number of network devices and different network topologies. As shown, the runtime heavily depends on the network topology. Tree topologies enable more provers to perform computations and communication simultaneously in the network. In fact, whereas more than one million prover devices arranged in a tree topology can be attested in less than 1.0 s, the attestation of 10000 provers in a chain topology requires 192.1 s. Fortunately, devices are in typical application scenarios much more likely connected in some form of tree topology than in long chains.

Furthermore, we also investigated the performance in quasi dynamic networks, in which the network topology changes in random ways before the token generation protocol is executed. In completely static networks, the initiator device can assume a static set of participating provers, such that provers do not have to transmit their identifiers during token generation. In quasi dynamic networks, which are represented as dotted lines in Figure 7.4, this is not possible, since



(a) Devices compute the aggregated public key of provers ad-hoc.



(b) Devices use a stored aggregated public key of provers.

Figure 7.5: Token exchange runtimes in various static topologies and with a varying ratio between prover and verifier devices.

provers may leave or join the network between runs of the token generation protocol. Due to the additional communication of device identifiers, the token generation runtime is slightly higher in quasi dynamic networks and increases with an increasing number of provers.

Token Exchange. Figure 7.5 illustrates the runtime required to exchange a single token between all devices in the network. The exchanged token was generated by all provers and attests their integrity. We varied the total number of devices, network topology, and ratio between prover and verifier devices. In addition, devices either computed the aggregated public key of provers ad-hoc, depicted in Figure 7.5a, or used a precomputed and stored aggregated public key, depicted in Figure 7.5b. During token exchange, the aggregated public key is required to verify the token signature, which protects the token integrity. The key must be computed whenever a token is received from an untrustworthy device and the token lists a new set of prover devices for which the receiver does not yet store the aggregated public key. As shown, the difference between computing the key ad-hoc and using a precomputed key is huge. Whereas in the first case the

runtime to exchange and verify a token in a network with one million devices amounts to a few minutes, in the latter case it is less than 0.7 s. However, we would like to emphasize that in networks with up to 4000 devices, the token exchange runtime is even in the worst-case, that is, with new sets of provers in each token exchange, always below 1 s.

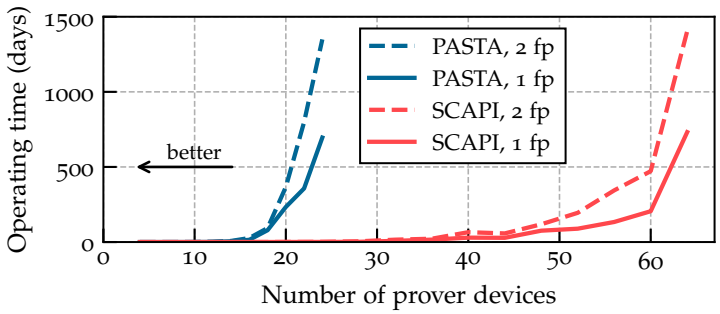
Another essential factor for the performance of the token exchange is the ratio between provers and verifiers in the network. In case the network contains many provers and only few untrustworthy devices, devices need to verify the signature of received tokens less often and the runtime decreases, as shown by the solid lines in Figure 7.5a. This is because devices can omit verifying the signature of tokens that are received from healthy provers (Section 7.3.3). On the contrary, if there are only few provers and many verifiers, devices need to verify tokens frequently. Nonetheless, in this case, each token contains less prover devices and the aggregated public key required to verify the token can be computed much faster. In total, this also results in a low runtime, as shown by the dashed lines. A balanced number of healthy provers and verifiers results in the highest runtime. Yet, when devices store the precomputed public key required to verify a received token, the impact of the ratio between provers and verifiers is less significant because verifying tokens is then much faster, as shown in Figure 7.5b.

Conclusion. We showed that PASTA is scalable to very large networks due to its small communication and computational overhead. In the best case, which is a static network with a uniform tree topology and only prover devices, one million provers can attest and verify the integrity of each other in less than 2.91 s (token generation plus token exchange). Yet, even in the worst case, which is a network with approximately 41 % verifier devices and a topology that changes between runs of the protocol, 1000 devices (410 verifiers and 590 provers) are able to verify the integrity of all 590 provers within 71.7 s.

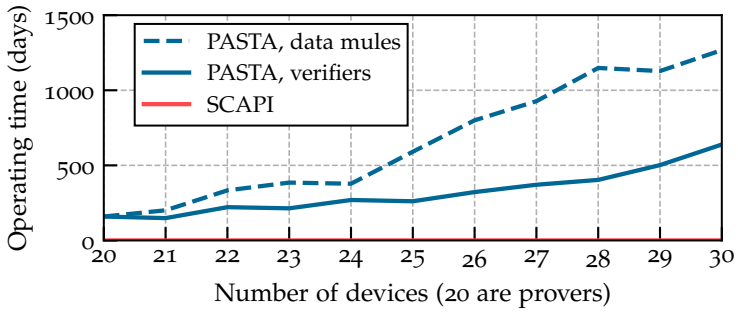
7.4.3 *Dynamic Network Simulations*

Simulation Setup. In order to simulate dynamic and disruptive network topologies, we extended our simulation setup for static networks (Section 7.4.2). Instead of using static connections between devices, we set the device communication range to 50 m and deployed devices randomly in a 1000m x 1000m area. During simulation, devices repeatedly select a random destination within the area and then move towards this destination at a specified speed, which is repeatedly set to a random value between 5 and 15 m/s (random waypoint mobility model). PASTA's parameters δ_{gen} and δ_{join} to initiate and join a token generation were set to 10 s and 5 s. In addition, we assumed that Adv_{hw} requires at least 10 min to physically tamper with a prover, i.e., $\delta_a = 10$ min. Thus, the simulation settings and parameters are the same as in the evaluation of SCAPI (Section 6.4).

Robustness of Attestation. To assess the robustness of PASTA, we investigated the runtime until prover devices are mistakenly regarded as physically compro-



(a) Varying number of false positives (fp).



(b) Single false positive (1 fp) and increasing number of non-prover devices.

Figure 7.6: Error-free runtime of prover attestation, i.e., runtime until prover devices are mistakenly regarded as physically compromised.

mised. In dynamic and disruptive networks, provers only occasionally have a connection to each other. Hence, a healthy prover may be unable to communicate with other provers for longer than δ_a time, whereupon the prover is regarded as physically compromised. Note that the considered scenarios deliberately go beyond typical application scenarios for autonomous embedded systems to determine the boundaries of PASTA.

Figure 7.6a shows the runtime until either one or two provers are falsely regarded as physically compromised. The network only consists of prover devices, whose number we varied. Furthermore, we set the concurrency factor β, which defines the number of provers Adv_{hw} can physically compromise within δ_a time, to ∞. Thus, we assumed that Adv_{hw} is able to physically attack all provers concurrently. In addition, we compared PASTA with SCAPI. As shown, the robustness of both protocols exponentially increases with the number of provers in the network. With more provers, the network becomes denser, which allows for more communication between the devices. By contrast, in sparser networks, chances are higher that a particular prover does not encounter any other prover within δ_a time, so that all other provers will regard the isolated prover as absent, hence, physically compromised. With the random movement of devices, it is significantly more likely that a single prover is isolated, as opposed to a group of provers. This is why the runtime for a misclassification of two provers is

on average almost twice as high as for a single prover. In comparison, SCAPI requires roughly three times more provers to achieve the same error-free runtime as PASTA, or, with the same number of provers, operates two orders of magnitude less robustly. This is because in SCAPI each prover must have a connection to other provers at least every $\delta_a/2$ time, as opposed to δ_a time in PASTA. Therefore, PASTA is significantly more robust to network dynamics and disruptions.

In practice, autonomous embedded systems usually not only contain provers, but also (potentially untrustworthy) devices that are not attested. To evaluate SCAPI and PASTA in these scenarios, we set the number of provers to 20 and then added an increasing number of either data mules (dashed lines) or verifiers (solid lines). Whereas verifiers continuously verify the integrity of provers, data mules (e.g., access points) merely store and forward tokens. As depicted in Figure 7.6b, PASTA runs much more robustly with an increasing number of non-prover devices. This is due to the fact that PASTA enables any device to propagate tokens in the network. By contrast, non-prover devices have no impact on the robustness of SCAPI. In SCAPI, provers communicate encrypted with a secret key that cannot be shared with (potentially untrustworthy) non-prover devices, which are therefore unable to participate in the attestation protocol. Figure 7.6b also shows that the error-free runtime of PASTA is higher in networks with data mules than with verifiers. The reason for this is that data mules do not verify the integrity of provers, so that fewer devices in the network can falsely classify a healthy prover as compromised.

In the next simulation, whose results are shown in Figure 7.7, we arranged an impassable horizontal barrier in the middle of a 800m x 800m area, and deployed half of all devices in each of the two partitions. As a result, connections between devices from the same partition are much more likely to occur than connections between devices from different partitions. In addition, we varied the number of provers and the concurrency factor β. If β is set to ∞, Adv_{hw} can compromise all provers concurrently, as in all previous simulations. With the barrier, it is more likely that whole groups of provers are separated from each other for longer than δ_a time, than single provers. This allows PASTA to make use of its unique feature to reunite separated groups of prover devices. A lower concurrency factor β allows separated groups of provers to be smaller and be separated from other groups for longer times. In fact, setting a slightly lower β in our simulations leads to an increased robustness of up to multiple orders of magnitude. In contrast, SCAPI immediately produces false positives after $\delta_a/2$ time, i.e., 5 min, with up to 90 provers (not shown in Figure 7.7). This is no surprise, since Figure 7.6a already showed that SCAPI performs significantly worse than PASTA with $\beta = \infty$.

Conclusion. We demonstrated that PASTA is significantly more robust to network disruptions than SCAPI (Section 6). In the same networks, PASTA has shown to achieve an average error-free operating time that is up to 450 times higher than SCAPI. This huge gap in the robustness even further increases when (i) the network contains additional non-prover devices, e.g., verifier or

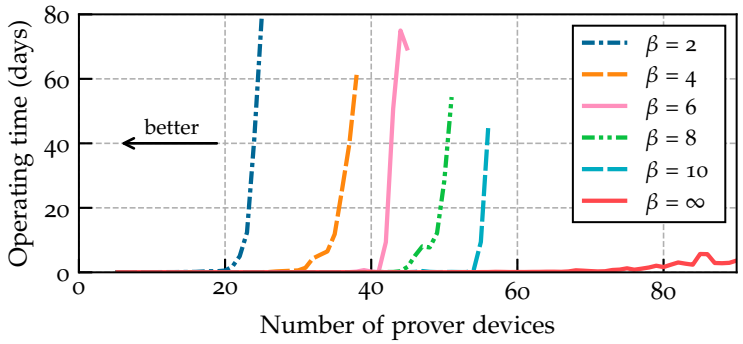


Figure 7.7: Error-free runtime of prover attestation with an impassable barrier in the deployment area that separates the network in two parts.

network infrastructure devices, or (ii) it is assumed that Adv_{hw} can only physically tamper with a limited number of provers simultaneously. Whereas in the first case, PASTA outperforms SCAPI by three orders of magnitude, the second case enables PASTA to run reliable in network topologies in which an attestation with SCAPI is impossible.

7.5 SUMMARY

In this chapter, we presented PASTA, an attestation protocol that is particularly suited for autonomous networks of embedded devices. As opposed to other protocols, PASTA (i) allows many provers to attest their integrity towards many verifiers in a scalable and efficient way, (ii) is decentralized, hence, independent of any entity that manages the attestation, and (iii) is able to detect physical attacks in a much more robust way than any existing protocol. In simulations based on real-world measurements, we showed that one million low-end embedded prover devices are able to attest their software and hardware integrity within 0.5 s in a token of only 68 bytes. The token can be verified by a low-end embedded device within 0.06 ms or 21 ms, depending on whether the token is received from a prover or a potentially untrustworthy network device. Furthermore, we demonstrated that PASTA is much more robust than our previously proposed attestation protocol SCAPI (Section 6). In disruptive networks, it achieves an error-free operating time that is several orders of magnitude higher than SCAPI.

8 CONCLUSION

We conclude this thesis by summarizing our contributions (Section 8.1) and outlining directions for future research (Section 8.2).

8.1 SUMMARY

Remote attestation constitutes an effective technique to face the increasing number of attacks on networked embedded systems. It allows to detect whether remote devices are in a trustworthy system state (or not). Thus, remote attestation enables to quickly identify and respond to attacks on embedded devices, and mitigate their damage. In this thesis, we advanced the state of the art in the attestation of embedded systems in three main ways.

First, we targeted the secure attestation of commodity low-end embedded devices. Existing attestation protocols rely on secure hardware to achieve strong security guarantees. The necessary secure hardware is, however, unavailable in low-end embedded devices. We presented ALE, an attestation protocol that relies on less secure hardware while providing the same (strong) security guarantees as existing hardware-based protocols. The relaxed requirements on secure hardware allow ALE to be in particular applicable to commodity low-end embedded devices. As a result, our protocol enables various novel applications that were previously impractical due to weak security guarantees or lack of secure hardware. To demonstrate this, we applied ALE in two specific use cases, namely, ECUs in road vehicles and code updates for wireless sensor networks. In the first use case, we proposed a scheme that ensures the secure and safe operation of embedded systems in road vehicles. We showed that our scheme is applicable to many automotive embedded systems, as they often adhere to automotive standards that fulfill ALE's demands on secure hardware. In the second use case, we proposed a secure code update scheme for mesh-networked low-end embedded devices. Our scheme enforces the proper installation of updates and the erasure of potential malware on all devices in the network. Unlike other solutions, our scheme is, due to its strong security guarantees, applicable in mesh networks, where an adversary may have compromised various devices.

Next, we aimed at the efficient attestation of multiple devices that are connected in particularly challenging networks. More specifically, we focused on highly dynamic and disruptive network topologies. In these topologies, existing attestation protocols are impractical or even inapplicable. We proposed SALAD, an attestation protocol that can efficiently verify multiple devices in highly dynamic and disruptive networks. In addition, SALAD provides an increased resilience against DoS attacks, allows a verifier to obtain the attestation result

from any device, and considers a physical adversary who is able to tamper with the hardware of devices in the network. In the presence of a physical adversary, SALAD guarantees a secure attestation of all physically uncompromised devices, unlike other collective attestation protocols. Nevertheless, SALAD is unable to detect physically compromised devices.

Finally, we explored practical attestation protocols that defend against invasive physical attacks. We presented two protocols, SCAPI and PASTA, that not only detect devices whose software is compromised but also devices whose hardware has been tampered with. Existing attestation protocols that aim to detect physically compromised devices suffer from limited scalability and robustness. Our first protocol, SCAPI, specifically addresses scalability and efficiency, in which respect it outperforms existing protocols by several orders of magnitudes. Furthermore, SCAPI entails only a low computational and memory overhead, which makes it applicable to low-end embedded devices. In comparison, our second protocol, PASTA, targets more powerful low-end and mid-range embedded devices. In return, PASTA provides more robustness against device outages and network disruptions than SCAPI. Additionally, PASTA allows embedded prover devices to attest their integrity towards multiple potentially untrustworthy embedded verifier devices. These properties make PASTA particularly suited for autonomous networks of embedded systems.

In short, we presented protocols that enable a more *secure*, *efficient*, and *robust* attestation of embedded devices. In addition, we *applied* one of our protocols in two novel usage scenarios. With our contributions, we advanced the practicality of remote attestation on embedded devices, which we hope will facilitate the deployment and use of remote attestation in practice.

8.2 FUTURE WORK

Although this thesis advanced remote attestation techniques for embedded systems in various directions, there are still open challenges that require future work. The following paragraphs outline these challenges.

Provable Secure Attestation. All our attestation protocols were carefully designed to meet the stated security goals (Section 2.4). Nevertheless, even most carefully designed protocols sometimes turn out to have security vulnerabilities. In order to provide more evidence for the security of attestation protocols, recent works aimed at provable security [10, 110, 111]. Thus far, VRASED [111] constitutes the only formally verified hardware-based attestation protocol. However, VRASED only considers the attestation of a single device and relies on hardware modifications that prevent its application in commodity embedded systems. Therefore, more research is needed to prove the security of contemporary collective attestation protocols or attestation protocols that are applicable to commodity embedded devices, which we presented in this thesis.

Runtime Attestation of Road Vehicles. In Section 4.1, we presented a scheme that ensures the secure and safe operation of embedded systems in road vehicles. Our scheme detects compromised automotive embedded systems at the start of the vehicle, but is unable to detect compromised devices after the vehicle is started. To also detect attacks during the operation of vehicles, our scheme can be extended with DIAT [4], a runtime attestation protocol for autonomous embedded systems. Nevertheless, besides detection, the incident response approach of our scheme must also be adjusted. So far, our scheme locks the ignition switch or power supply whenever a compromised state has been detected. However, during operation, a vehicle may be moving at high speed along a highway. In this case, shutting down the engine or power supply threatens the safety of vehicles, vehicle occupants, and other road users. Hence, more research is needed to find appropriate incident responses for this application scenario.

Analysis of Physical Tamper Resistance. Another interesting direction of research is to investigate the resistance of commodity embedded systems against invasive and semi-invasive physical attacks. Existing works typically explore novel physical attacks but lack precise numbers about the costs and time that is needed to perform these attacks. By quantifying the precise resistance against physical attacks, the capabilities of a physical adversary can be narrowed down and parameters for the absence detection can be adjusted accordingly. As a result, this would enhance the security and robustness of our proposed attestation protocols that defend against physical attacks (Chapters 6 and 7).

Extended Evaluation. To evaluate our attestation protocols, we implemented them, measured their performance in small networks, and then simulated large networks based on our measurements. However, network simulations always introduce modeling and discretization errors. One way towards more realistic simulation results is to use non-synthetic mobility trace, e.g., recorded from actual deployment scenarios. Nonetheless, to obtain precise evaluation results for specific scenarios, our protocols would need to be tested in large-scale real-world experiments. Ideally, the protocols are evaluated in realistic application scenarios, such as drone-based delivery systems or wireless sensor networks. Unfortunately, performing such real-world experiments involves a significant effort. This is because different network topologies and device movements need to be considered and a large testbed with hundreds of interconnected embedded devices is required.

BIBLIOGRAPHY

[1] ARM Holdings. *mbed TLS*. URL: https://tls.mbed.org/.

[2] AUTOSAR. *Specification of Secure Onboard Communication*. V.4.3.1.

[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 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM. 2016, pp. 743–754.

[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: *26th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society. 2019.

[5] Trusted Computing Platform Alliance. *Main Specification, Version 1.1 b*. 2002.

[6] Tiago Alves and Don Felton. *TrustZone: Integrated Hardware and Software Security*. ARM Technical White paper. 2004.

[7] Moreno Ambrosin, Mauro Conti, Ahmad Ibrahim, Gregory Neven, Ahmad-Reza Sadeghi, and Matthias Schunter. "SANA: Secure and Scalable Aggregate Network Attestation." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM. 2016, pp. 731–742.

[8] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. "Innovative technology for CPU based attestation and sealing." In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM. 2013.

[9] William A Arbaugh, David J Farber, and Jonathan M Smith. "A secure and reliable bootstrap architecture." In: *1997 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 1997, pp. 65–71.

[10] 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)*. ACM. 2013, pp. 1–12.

[11] N Asokan, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter, Gene Tsudik, and Christian Wachsmann. "SEDA: Scalable Embedded Device Attestation." In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM. 2015, pp. 964–975.

[12] 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*. Vol. 37. 11. 2018, pp. 2290–2300.

[13] Atmel. *Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V Datasheet*. 2014.

[14] Alexander Becher, Zinaida Benenson, and Maximillian Dornseif. "Tampering with motes: Real-world physical attacks on wireless sensor networks." In: *Third International Conference on Security in Pervasive Computing (SPC)*. Vol. 3934. Lecture Notes in Computer Science. Springer, 2006, pp. 104–118.

[15] Daniel Beer. *Curve25519 and Ed25519 for low-memory systems*. 2014. URL: https://www.dlbeer.co.nz/oss/c25519.html.

[16] Mihir Bellare and Gregory Neven. "Multi-signatures in the plain public-key model and a general forking lemma." In: *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*. ACM. 2006, pp. 390–399.

[17] Daniel J Bernstein. *Supercop: System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives*. URL: https://bench.cr.yp.to/supercop.html.

[18] Daniel J Bernstein. "Curve25519: new Diffie-Hellman speed records." In: *9th International Conference on Theory and Practice of Public-Key Cryptography (PKC)*. Vol. 3958. Lecture Notes in Computer Science. Springer, 2006, pp. 207–228.

[19] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. "High-speed high-security signatures." In: *Journal of Cryptographic Engineering*. Vol. 2. 2. Springer, 2012, pp. 77–89.

[20] Christian Bettstetter, Giovanni Resta, and Paolo Santi. "The Node Distribution of the Random Waypoint Mobility Model for Wireless Ad Hoc Networks." In: *IEEE Transactions on Mobile Computing (TMC)*. Vol. 2. 3. 2003, pp. 257–269.

[21] Ben Romdhanne Bilel, Nikaein Navid, and Mohamed Said Mosli Bouksiaa. "Hybrid cpu-gpu distributed framework for large scale mobile networks simulation." In: *16th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE Computer Society, 2012, pp. 44–53.

[22] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. "Aggregate and verifiably encrypted signatures from bilinear maps." In: *Advances in Cryptology - International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Vol. 2656. Lecture Notes in Computer Science. Springer, 2003, pp. 416–432.

[23] 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, pp. 341–346.

[24] Coen Bron and Joep Kerbosch. "Algorithm 457: finding all cliques of an undirected graph." In: *Communications of the ACM*. Vol. 16. 9. ACM, 1973, pp. 575–576.

[25] T Ryan Burchfield, S Venkatesan, and Douglas Weiner. "Maximizing throughput in zigbee wireless networks through analysis, simulations and implementations." In: *Proceedings of the 1st International Workshop on Localized Algorithms and Protocols for Wireless Sensor Networks (LOCALGOS)*. 2007, pp. 15–29.

[26] Jenna Burrell, Tim Brooke, and Richard Beckwith. "Vineyard computing: Sensor networks in agricultural production." In: *IEEE Pervasive computing*. Vol. 3. 1. 2004, pp. 38–45.

[27] John Butterworth, Corey Kallenberg, Xeno Kovah, and Amy Herzog. "Bios chronomancy: Fixing the core root of trust for measurement." In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM. 2013, pp. 25–36.

[28] Xavier Carpent, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. "Lightweight Swarm Attestation: a Tale of Two LISA-s." In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIACCS)*. ACM. 2017, pp. 86–100.

[29] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. "On the difficulty of software-based attestation of embedded devices." In: *Proceedings of the 2009 ACM Conference on Computer and Communications Security (CCS)*. ACM. 2009, pp. 400–409.

[30] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. "Comprehensive experimental analyses of automotive attack surfaces." In: *Proceedings of the 20th USENIX Security Symposium (USENIX-Security)*. USENIX Association, 2011, pp. 77–92.

[31] Jiming Chen, Xianghui Cao, Peng Cheng, Yang Xiao, and Youxian Sun. "Distributed collaborative control for industrial automation with wireless sensor and actuator networks." In: *IEEE Transactions on Industrial Electronics*. Vol. 57. 12. 2010, pp. 4219–4230.

[32] Mauro Conti, Roberto Di Pietro, Luigi Vincenzo Mancini, and Alessandro Mei. "Emergent properties: detection of the node-capture attack in mobile wireless sensor networks." In: *Proceedings of the First ACM Conference on Wireless Network Security (WiSec)*. ACM. 2008, pp. 214–219.

[33] Mauro Conti, Roberto Di Pietro, Andrea Gabrielli, Luigi V Mancini, and Alessandro Mei. "The smallville effect: social ties make mobile networks more secure against node capture attack." In: *Proceedings of the 8th ACM International Workshop on Mobility Management & Wireless Access (MOBIWAC)*. ACM, 2010, pp. 99–106.

[34] Andrei Costin, Jonas Zaddach, Aurélien Francillon, Davide Balzarotti, and Sophia Antipolis. "A large-scale analysis of the security of embedded firmwares." In: *Proceedings of the 23rd USENIX Security Symposium (USENIX-Security)*. USENIX Association, 2014, pp. 95–110.

[35] Payment Card Industry Security Standards Council. *Payment Card Industry PTS HSM Security Requirements v2.0*. PCI, Wakefield, MA, USA, 2012.

[36] Ang Cui, Michael Costello, and Salvatore J. Stolfo. "When Firmware Modifications Attack: A Case Study of Embedded Exploitation." In: *20th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society. 2013.

[37] Ruan De Clercq, Leif Uhsadel, Anthony Van Herrewege, and Ingrid Verbauwhede. "Ultra low-power implementation of ECC on the ARM Cortex-M0+." In: *Proceedings of the 51st Annual Design Automation Conference (DAC)*. ACM, 2014, pp. 1121–1126.

[38] 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, pp. 241–246.

[39] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. "LiteHAX: lightweight hardware-assisted attestation of program execution." In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2018, pp. 1–8.

[40] Danny Dolev and Andrew Yao. "On the security of public key protocols." In: *IEEE Transactions on Information Theory*. Vol. 29. 2. IEEE, 1983, pp. 198–208.

[41] Wei Dong, Chun Chen, Jiajun Bu, and Wen Liu. "Optimizing Relocatable Code for Efficient Software Update in Networked Embedded Systems." In: *ACM Transactions on Sensor Networks (TOSN)*. Vol. 11. 2. ACM, 2014, 22:1–22:34.

[42] Michael Düll, Björn Haase, Gesine Hinterwälder, Michael Hutter, Christof Paar, Ana Helena Sánchez, and Peter Schwabe. "High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers." In: *Designs, Codes and Cryptography*. Vol. 77. 2-3. Springer, 2015, pp. 493–514.

[43] Rachid El Bansarkhani and Jan Sturm. "An Efficient Lattice-Based Multisignature Scheme with Applications to Bitcoins." In: *15th International Conference on Cryptology and Network Security (CANS)*. Vol. 10052. Lecture Notes in Computer Science. Springer, 2016, pp. 140–155.

[44] 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)*. ACM. 2017, pp. 99–110.

[45] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. "SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust." In: *19th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society. 2012.

[46] R Escherich, I Ledendecker, C Schmal, B Kuhls, C Grothe, and F Scharberth. *SHE – Secure Hardware Extension Functional Specification Version 1.1 (rev 439)*. 2009.

[47] Arved Esser, Florian Kohnhäuser, Nadine Ostern, Kevin Engleson, and Stephan Rinderknecht. "Enabling a Privacy-Preserving Synthesis of Representative Driving Cycles from Fleet Data using Data Aggregation." In: *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. IEEE Computer Society, 2018, pp. 1384–1389.

[48] Donald L Evans, Phillip Bond, and Arden Bement. *FIPS Pub 140-2: Security Requirements for Cryptographic Modules*. Federal Information Processing Standards Publication, Mar. 2002.

[49] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. "Komodo: Using verification to disentangle secure-enclave hardware from software." In: *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. ACM, 2017, pp. 287–305.

[50] Ian Foster, Andrew Prudhomme, Karl Koscher, and Stefan Savage. "Fast and Vulnerable: A Story of Telematic Failures." In: *9th USENIX Workshop on Offensive Technologies (WOOT)*. USENIX Association, 2015.

[51] Aurélien Francillon, Quan Nguyen, Kasper B Rasmussen, and Gene Tsudik. "A minimalist approach to remote attestation." In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. European Design and Automation Association, 2014, pp. 1–6.

[52] Freescale Semiconductor. *Using the Kinetis Flash Execute Only Access Control Feature - 6.3 Entry into execute-only code on the ARM Cortex-M4 core*. Application Note. 2015.

[53] Walter Fuertes, Diego Carrera, César Villacís, Theofilos Toulkeridis, Fernando Galárraga, Edgar Torres, and Hernán Aules. "Distributed system as internet of things for a new low-cost, air pollution wireless monitoring on real time." In: *19th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*. IEEE Computer Society, 2015, pp. 58–67.

[54] Gartner Inc. *Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016*. 2017. URL: http://www.gartner.com/newsroom/id/3598917.

[55] Gartner Inc. *Gartner Says Detection and Response is Top Security Priority for Organizations in 2017*. 2017. URL: http://www.gartner.com/newsroom/id/3638017.

[56] GitHub: bitcoin-core/secp256k1. "Optimized C library for EC operations on curve secp256k1." In: URL: https://github.com/bitcoin-core/secp256k1.

[57] Zheng Guo, Gioele Colombi, Bing Wang, Jun-Hong Cui, Dario Maggiorini, and Gian Paolo Rossi. "Adaptive routing in underwater delay/disruption tolerant sensor networks." In: *2008 Fifth Annual Conference on Wireless on Demand Network Systems and Services (WONS)*. IEEE Computer Society, 2008, pp. 31–39.

[58] Andrew Hagedorn, David Starobinski, and Ari Trachtenberg. "Rateless deluge: Over-the-air programming of wireless sensor networks using random linear codes." In: *Proceedings of the 7th International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE Computer Society, 2008, pp. 457–466.

[59] Steven Hanna, Rolf Rolles, Andrés Molina-Markham, Pongsin Poosankam, Kevin Fu, and Dawn Song. "Take two software updates and see me in the morning: The case for software security evaluations of medical devices." In: *2nd USENIX Workshop on Health Security and Privacy (HealthSec)*. USENIX Association, 2011.

[60] Daojing He, Chun Chen, Sammy Chan, and Jiajun Bu. "SDRP: A secure and distributed reprogramming protocol for wireless sensor networks." In: *IEEE Transactions on Industrial Electronics*. Vol. 59. 11. 2012, pp. 4155–4163.

[61] Kevin Hemsley and Ronald Fisher. "A History of Cyber Incidents and Threats Involving Industrial Control Systems." In: *12th IFIP WG 11.10 International Conference on Critical Infrastructure Protection (ICCIP)*. Vol. 542. IFIP Advances in Information and Communication Technology. Springer, 2018, pp. 215–242.

[62] Jun-Won Ho. "Distributed detection of node capture attacks in wireless sensor networks." In: *Smart Wireless Sensor Networks*. ISBN: 978-953-307-261-6. InTech, 2010, pp. 345–360.

[63] Lingxuan Hu and David Evans. "Secure aggregation for wireless networks." In: *2003 Symposium on Applications and the Internet Workshops (SAINT)*. IEEE Computer Society, 2003, pp. 384–391.

[64] *INET Framework*. URL: https://inet.omnetpp.org/.

[65] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Gene Tsudik. "Us-aid: Unattended scalable attestation of iot devices." In: *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*. IEEE Computer Society. 2018, pp. 21–30.

[66] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Gene Tsudik. "HEALED: HEaling & Attestation for Low-end Embedded Devices." In: *23rd International Conference on Financial Cryptography and Data Security (FC)*. Lecture Notes in Computer Science. Springer, 2019.

[67] 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)*. ACM. 2017, pp. 64–74.

[68] Ahmad Ibrahim, Ahmad-Reza Sadeghi, Gene Tsudik, and Shaza Zeitouni. "DARPA: Device attestation resilient to physical attacks." In: *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*. ACM. 2016, pp. 171–182.

[69] Intel Open Source Technology Center. *TinyCrypt Cryptographic Library*. 2017. URL: https://github.com/01org/tinycrypt.

[70] Kazuharu Itakura. "A public-key cryptosystem suitable for digital multisignatures." In: *NEC J. Res. Dev.* Vol. 71. 1983.

[71] Lukas Jäger, Richard Petri, and Andreas Fuchs. "Rolling dice: Lightweight remote attestation for cots iot hardware." In: *Proceedings of the 12th International Conference on Availability, Reliability and Security (ARES)*. ACM, 2017, pp. 951–958.

[72] Reiner Jedermann, Christian Behrens, Detmar Westphal, and Walter Lang. "Applying autonomous sensor systems in logistics - Combining sensor networks, RFIDs and software agents." In: *Sensors and Actuators A: Physical*. Vol. 132. 1. Elsevier, 2006, pp. 370–375.

[73] Ik Rae Jeong, Jonathan Katz, and Dong Hoon Lee. "One-round protocols for two-party authenticated key exchange." In: *Second International Conference on Applied Cryptography and Network Security (ACNS)*. Vol. 3089. Lecture Notes in Computer Science. Springer, 2004, pp. 220–232.

[74] Jiong Jin, Jayavardhana Gubbi, Slaven Marusic, and Marimuthu Palaniswami. "An information framework for creating a smart city through internet of things." In: *IEEE Internet of Things Journal*. Vol. 1. 2. 2014, pp. 112–121.

[75] Ghassan O Karame and Wenting Li. "Secure Erasure and Code Update in Legacy Sensors." In: *8th International Conference on Trust and Trustworthy Computing (TRUST)*. Vol. 9229. Lecture Notes in Computer Science. Springer, 2015, pp. 283–299.

[76] Kaspersky Lab. *New trends in the world of IoT threats*. 2018. URL: https://securelist.com/new-trends-in-the-world-of-iot-threats/87991/.

[77] Jonathan Katz and Andrew Lindell. "Aggregate message authentication codes." In: *The Cryptographers' Track at the RSA Conference (CT-RSA)*. Vol. 4964. Lecture Notes in Computer Science. Springer, 2008, pp. 155–169.

[78] Stefan Katzenbeisser, Ünal Kocabaş, Vladimir Rožić, Ahmad-Reza Sadeghi, Ingrid Verbauwhede, and Christian Wachsmann. "PUFs: Myth, fact or busted? A security evaluation of physically unclonable functions (PUFs) cast in silicon." In: *14th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Vol. 7428. Lecture Notes in Computer Science. Springer, 2012, pp. 283–301.

[79] Ari Keränen, Mehmet Ersue, and Carsten Bormann. *Terminology for Constrained-Node Networks*. RFC 7228. Internet Engineering Task Force (IETF), 2014.

[80] Ari Keränen, Jörg Ott, and Teemu Kärkkäinen. "The ONE simulator for DTN protocol evaluation." In: *Proceedings of the 2nd International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUtools)*. 55. ICST/ACM, 2009.

[81] Wolfgang Killmann and Kerstin Lemke-Rust. *Common Criteria Protection Profile–Cryptographic Modules, Security Level "Enhanced"*. Bundesamt für Sicherheit in der Informationstechnik (BSI). 2008.

[82] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. "TrustLite: a security architecture for tiny embedded devices." In: *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*. ACM, 2014, pp. 1001–1014.

[83] Florian Kohnhäuser, Niklas Büscher, and Stefan Katzenbeisser. "SALAD: Secure and Lightweight Attestation of Highly Dynamic and Disruptive Networks." In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security (ASIACCS)*. ACM. 2018, pp. 329–342.

[84] Florian Kohnhäuser, Niklas Büscher, and Stefan Katzenbeisser. "A Practical Attestation Protocol for Autonomous Embedded Systems." In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE Computer Society, 2019.

[85] Florian Kohnhäuser and Stefan Katzenbeisser. "Secure Code Updates for Mesh Networked Commodity Low-End Embedded Devices." In: *21st European Symposium on Research in Computer Security (ESORICS)*. Vol. 9879. Lecture Notes in Computer Science. Springer, 2016, pp. 320–338.

[86] Florian Kohnhäuser, Dominik Püllen, and Stefan Katzenbeisser. "Ensuring the Safe and Secure Operation of Electronic Control Units in Road Vehicles." In: *2019 IEEE Security and Privacy Workshops (SPW)*. IEEE Computer Society, 2019.

[87] Florian Kohnhäuser, André Schaller, and Stefan Katzenbeisser. "PUF-based software protection for low-end embedded devices." In: *8th International Conference on Trust and Trustworthy Computing (TRUST)*. Vol. 9229. Lecture Notes in Computer Science. Springer, 2015, pp. 3–21.

[88] Florian Kohnhäuser, Niklas Büscher, Sebastian Gabmeyer, and Stefan Katzenbeisser. "SCAPI: A Scalable Attestation Protocol to Detect Software and Physical Attacks." In: *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*. ACM. 2017, pp. 75–86.

[89] Florian Kohnhäuser, Milan Stute, Lars Baumgärtner, Lars Almon, Stefan Katzenbeisser, Matthias Hollick, and Bernd Freisleben. "SEDCOS: A Secure Device-to-Device Communication System for Disaster Scenarios." In: *2017 IEEE 42nd Conference on Local Computer Networks (LCN)*. IEEE Computer Society, 2017, pp. 195–198.

[90] Markus Kosmal. *SharedAES-GCM: Attempt for a cross platform AES-GCM encryption*. URL: https://mko-x.github.io/SharedAES-GCM/.

[91] Xeno Kovah, Corey Kallenberg, Chris Weathers, Alexander Herzog, Matthew Albin, and John Butterworth. "New results for timing-based attestation." In: *2012 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2012, pp. 239–253.

[92] Hugo Krawczyk, Ran Canetti, and Mihir Bellare. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. Network Working Group, 1997.

[93] Sandeep Kulkarni and Limin Wang. "Energy-efficient multihop reprogramming for sensor networks." In: *ACM Transactions on Sensor Networks (TOSN)*. Vol. 5. 2. ACM, 2009, pp. 1601–1640.

[94] James LaPiedra. *The Information Security Process: Prevention, Detection and Response*. SANS Institute: Information Security Reading Room 20, 2002.

[95] Laurie Law, Alfred Menezes, Minghua Qu, Jerry Solinas, and Scott Vanstone. "An efficient protocol for authenticated key agreement." In: *Designs, Codes and Cryptography*. Vol. 28. 2. Springer, 2003, pp. 119–134.

[96] Yee Wei Law, Yu Zhang, Jiong Jin, Marimuthu Palaniswami, and Paul Havinga. "Secure rateless deluge: Pollution-resistant reprogramming and data dissemination for wireless sensor networks." In: *EURASIP Journal on Wireless Communications and Networking*. Vol. 2011. Hindawi Publishing Corp., 2011.

[97] Gyesik Lee, Hisashi Oguma, Akira Yoshioka, Rie Shigetomi, Akira Otsuka, and Hideki Imai. "Formally verifiable features in embedded vehicular security systems." In: *IEEE Vehicular Networking Conference (VNC)*. IEEE Computer Society, 2009, pp. 1–7.

[98] Yanlin Li, Jonathan M McCune, and Adrian Perrig. "VIPER: verifying the integrity of PERipherals' firmware." In: *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*. ACM. 2011, pp. 3–16.

[99] Linaro. *OP-TEE: Open Portable Trusted Execution Environment*. URL: op-tee.org.

[100] Linux Foundation.. *Intel Quark Microcontroller Software Interface*. URL: https://downloadcenter.intel.com/download/25619/Intel-Quark-Microcontroller-Software-Interface.

[101] Linux Foundation.. *Zephyr Project*. URL: https://www.zephyrproject.org/.

[102] Steve Lu, Rafail Ostrovsky, Amit Sahai, Hovav Shacham, and Brent Waters. "Sequential Aggregate Signatures and Multisignatures Without Random Oracles." In: *Advances in Cryptology - International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Vol. 4004. Lecture Notes in Computer Science. Springer, 2006, pp. 465–485.

[103] Ken Mackay. *Micro-ECC*. URL: http://kmackay.ca/micro-ecc/.

[104] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. "Simple Schnorr Multi-Signatures with Applications to Bitcoin." In: *Designs, Codes and Cryptography*. Springer, 2018, pp. 1–26.

[105] Christian Meurisch, Julien Gedeon, Artur Gogel, The An Binh Nguyen, Fabian Kaup, Florian Kohnhaeuser, Lars Baumgaertner, Milan Schmittner, and Max Muehlhaeuser. "Temporal coverage analysis of router-based cloudlets using human mobility patterns." In: *2017 IEEE Global Communications Conference (GLOBECOM)*. IEEE Computer Society, 2017, pp. 1–6.

[106] Silvio Micali, Kazuo Ohta, and Leonid Reyzin. "Accountable-subgroup multisignatures." In: *Proceedings of the 8th ACM Conference on Computer and Communications Security (CCS)*. ACM. 2001, pp. 245–254.

[107] Charlie Miller and Chris Valasek. "Remote exploitation of an unaltered passenger vehicle." In: *Black Hat USA*. 2015.

[108] Thomas Morris. "Trusted platform module." In: *Encyclopedia of Cryptography and Security, 2nd Ed.* Springer, 2011, pp. 1332–1335.

[109] 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 22th USENIX Security Symposium (USENIX-Security)*. USENIX Association, 2013, pp. 479–494.

[110] Ivan Oliveira Nunes, Ghada Dessouky, Ahmad Ibrahim, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik. "Towards Systematic Design of Collective Remote Attestation Protocols." In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2019.

[111] Ivan 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)*. Santa Clara, CA, 2019.

[112] Stefan Nürnberger and Christian Rossow. "- vatiCAN - Vetted, Authenticated CAN Bus." In: *18th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Vol. 9813. Lecture Notes in Computer Science. Springer, 2016, pp. 106–124.

[113] Hisashi Oguma, Akira Yoshioka, Makoto Nishikawa, Rie Shigetomi, Akira Otsuka, and Hideki Imai. "New attestation based security architecture for in-vehicle communication." In: *Proceedings of the Global Communications Conference (GLOBECOM)*. IEEE, 2008, pp. 1909–1914.

[114] Haemin Park, Dongwon Seo, Heejo Lee, and Adrian Perrig. "SMATT: Smart Meter ATTestation Using Multiple Target Selection and Copy-Proof Memory." In: *Computer Science and its Applications*. Springer, 2012, pp. 875–887.

[115] Youngseok Park, Yunmok Son, Hocheol Shin, Dohyun Kim, and Yongdae Kim. "This Ain't Your Dose: Sensor Spoofing Attack on Medical Infusion Pump." In: *10th USENIX Workshop on Offensive Technologies (WOOT)*. USENIX Association, 2016.

[116] Daniele Perito and Gene Tsudik. "Secure Code Update for Embedded Devices via Proofs of Secure Erasure." In: *15th European Symposium on Research in Computer Security (ESORICS)*. Vol. 6345. Lecture Notes in Computer Science. Springer. 2010, pp. 643–662.

[117] Bartosz Przydatek, Dawn Song, and Adrian Perrig. "SIA: Secure information aggregation in sensor networks." In: *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (Sensys)*. ACM, 2003, pp. 255–265.

[118] Bing Qiao, Kecheng Liu, and Chris Guy. "A multi-agent system for building control." In: *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT)*. IEEE Computer Society, 2006, pp. 653–659.

[119] Andreea-Ina Radu and Flavio D Garcia. "LeiA: A lightweight authentication protocol for CAN." In: *21st European Symposium on Research in Computer Security (ESORICS)*. Vol. 9879. Lecture Notes in Computer Science. Springer, 2016, pp. 283–300.

[120] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, et al. "fTPM: A Software-Only Implementation of a TPM Chip." In: *Proceedings of the 25th USENIX Security Symposium (USENIX-Security)*. USENIX Association, 2016, pp. 841–856.

[121] Srivaths Ravi, Anand Raghunathan, and Srimat Chakradhar. "Tamper resistance mechanisms for secure embedded systems." In: *Proceedings of the 17th International Conference on VLSI Design (VLSID)*. IEEE Computer Society, 2004, pp. 605–611.

[122] Michele Rossi, Nicola Bui, Giovanni Zanca, Luca Stabellini, Riccardo Crepaldi, and Michele Zorzi. "SYNAPSE++: code dissemination in wireless sensor networks using fountain codes." In: *IEEE Transactions on Mobile Computing (TMC)*. Vol. 9. 12. IEEE, 2010, pp. 1749–1765.

[123] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert Van Doorn. "Design and Implementation of a TCG-based Integrity Measurement Architecture." In: *Proceedings of the 13th USENIX Security Symposium (USENIX-Security)*. USENIX Association, 2004, pp. 223–238.

[124] Dries Schellekens, Brecht Wyseur, and Bart Preneel. "Remote attestation on legacy operating systems with trusted platform modules." In: *Science of Computer Programming*. Vol. 74. 1-2. Elsevier, 2008, pp. 13–22.

[125] Claus-Peter Schnorr. "Efficient signature generation by smart cards." In: *Journal of cryptology*. Vol. 4. 3. Springer, 1991, pp. 161–174.

[126] Geert-Jan Schrijen and Vincent van der Leest. "Comparative analysis of SRAM memories used as PUF primitives." In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2012, pp. 1319–1324.

[127] Steffen Schulz, André Schaller, Florian Kohnhäuser, and Stefan Katzenbeisser. "Boot Attestation: Secure Remote Reporting with Off-The-Shelf IoT Sensors." In: *22nd European Symposium on Research in Computer Security (ESORICS)*. Vol. 10493. Lecture Notes in Computer Science. Springer, 2017, pp. 437–455.

[128] Leo Selavo, Anthony Wood, Qing Cao, Tamim Sookoor, Hengchang Liu, Aravind Srinivasan, Yafeng Wu, Woochul Kang, John Stankovic, Don Young, et al. "Luster: wireless sensor network for environmental research." In: *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems (Sensys)*. ACM, 2007, pp. 103–116.

[129] Arvind Seshadri, Mark Luk, and Adrian Perrig. "SAKE: Software attestation for key establishment in sensor networks." In: *4th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS)*. Vol. 5067. Lecture Notes in Computer Science. Springer, 2008, pp. 372–385.

[130] Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. "Swatt: Software-based attestation for embedded devices." In: *2004 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2004, pp. 272–282.

[131] 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 Symposium on Operating Systems Principles (SOSP)*. ACM. 2005, pp. 1–16.

[132] 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 (WISE)*. ACM. 2006, pp. 85–94.

[133] Sergei Petrovich Skorobogatov. "Semi-invasive attacks: a new approach to hardware security analysis." PhD thesis. University of Cambridge, 2005.

[134] Sergei Skorobogatov. "Physical attacks on tamper resistance: progress and lessons." In: *Proceedings of the 2nd ARO Special Workshop on Hardware Assurance, Washington, DC*. 2011.

[135] Sergei Skorobogatov. "Physical attacks and tamper resistance." In: *Introduction to Hardware Security and Trust*. Springer, 2012, pp. 143–173.

[136] Sergei Skorobogatov. "How microprobing can attack encrypted memory." In: *2017 Euromicro Conference on Digital System Design (DSD)*. IEEE. 2017, pp. 244–251.

[137] Paul Stelling, Cheryl DeMatteis, Ian Foster, Carl Kesselman, Craig Lee, and Gregor von Laszewski. "A fault detection service for wide area distributed computations." In: *Cluster Computing*. Vol. 2. 2. Springer, 1999, pp. 117–128.

[138] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. "Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning." In: *2016 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2016, pp. 526–545.

[139] Trusted Computing Group (TCG). *TPM Main Part 1 Design Principles, Specification Version 1.2*. 2006.

[140] Gelareh Taban and Virgil Gligor. "Efficient handling of adversary attacks in aggregation applications." In: *13th European Symposium on Research in Computer Security (ESORICS)*. Vol. 5283. Lecture Notes in Computer Science. Springer, 2008, pp. 66–81.

[141] Texas Instruments. *Stellaris LM4F120H5QR Microcontroller Data Sheet*. 2013.

[142] Texas Instruments. *Software IP Protection on MSP432P4xx Microcontrollers - 10.1 Interrupt Handling in IP Protected Secure Zone*. 2015.

[143] Noriki Uchida, Noritaka Kawamura, Tomoyuki Ishida, and Yoshitaka Shibata. "Proposal of autonomous flight wireless nodes with delay tolerant networks for disaster use." In: *Eighth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*. IEEE Computer Society, 2014, pp. 146–151.

[144] Osman Ugus, Dirk Westhoff, and Jens-Matthias Bohli. "A ROM-friendly secure code update mechanism for WSNs using a stateful-verifier τ-time signature scheme." In: *Proceedings of the Second ACM Conference on Wireless Network Security (WISEC)*. ACM. ACM, 2009, pp. 29–40.

[145] Jo Van Bulck, Jan Tobias Mühlberg, and Frank Piessens. "VulCAN: Efficient component authentication and software isolation for automotive control networks." In: *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)*. ACM, 2017, pp. 225–237.

[146] András Varga and Rudolf Hornig. "An overview of the OMNeT++ simulation environment." In: *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops (SIMUtools)*. 60. ICST/ACM, 2008.

[147] Peng Wang, Dengguo Feng, Changlu Lin, and Wenling Wu. "Security of Truncated MACs." In: *4th International Conference of Information Security and Cryptology (INSCRYPT)*. Vol. 5487. Lecture Notes in Computer Science. Springer, 2009, pp. 96–114.

[148] Marko Wolf and Timo Gendrullis. "Design, implementation, and evaluation of a vehicular hardware security module." In: *14th International Conference on Information Security and Cryptology (ICISC)*. Vol. 7259. Lecture Notes in Computer Science. Springer, 2011, pp. 302–318.

[149] Yi Yang, Xinran Wang, Sencun Zhu, and Guohong Cao. "SDAP: A secure hop-by-hop data aggregation protocol for sensor networks." In: *ACM Transactions on Information and System Security (TISSEC)*. Vol. 11. 4. ACM, 2008, pp. 1801–1843.

[150] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. "ATRIUM: Runtime attestation resilient under memory attacks." In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 384–391.