Secure Code Updates for Mesh Networked Commodity Low-End Embedded Devices

Florian Kohnhäuser, Stefan Katzenbeisser

TU Darmstadt, Security Engineering Group
lastname@seceng.informatik.tu-darmstadt.de

Abstract. Mesh networked low-end embedded devices are increasingly used in various scenarios, including industrial control, wireless sensing, robot swarm communication, or building automation. Recently, more and more software vulnerabilities in embedded systems are disclosed, as they become appealing targets for cyber attacks. In order to patch these systems, an efficient and secure code update mechanism is required. However, existing solutions are unable to provide verifiable code updates for networked commodity low-end embedded devices. This work presents a novel code update scheme which verifies and enforces the correct installation of code updates on all devices in the network. After update distribution and installation, devices mutually attest and verify each others' software state. Devices being in an untrustworthy state are excluded from the network. In this way, the scheme enforces software integrity as well as software up-to-dateness on all devices in the network. Issuing a secure code update, the network operator is able to learn the identity of all trustworthy and all untrustworthy devices. We demonstrate that the proposed scheme is applicable to a wide range of existing commodity low-end embedded systems. Furthermore, we show that the scheme is practically usable in networks with tens of thousands of devices.

1 Introduction

The continuous cost reduction and miniaturization of electronic devices commences a new technological revolution of omnipresent embedded devices. Trends like the Internet of Things, Smarter Planet, Industry 4.0, or Smart Cities aim at applying networked embedded systems in virtually every aspect of our life. Wireless technologies like IEEE 802.11s, IEEE 802.15.4, ZigBee, Z-Wave, or Bluetooth facilitate the establishment of large mesh networks consisting of numerous embedded systems. In a mesh network, all devices cooperate in the distribution of data in the network, forming a decentralized and self-organized network topology. Nowadays, wireless mesh networked embedded devices are already widely used in industrial control, wireless sensor networks, home automation, building automation, military communication, or community networks. These systems often perform security or safety-critical tasks, or process privacy-sensitive information. In addition, they commonly lack effective security mechanisms due to their low production costs as well as their small and simple system architecture.

These circumstances made them appealing targets for cyber attacks. Consequently, many software vulnerabilities in embedded systems have been revealed lately [11, 19, 35]. In order to fix such vulnerabilities, it is vital that low-end embedded devices provide secure code update mechanisms.

A secure code update scheme for the above described application 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 eventually be informed about the integrity of the software state of all devices in the network.

However, to the best of our knowledge, there is no solution which satisfies all these requirements. Software- and PoSE-based (Proofs of Secure Erasure) approaches are applicable to commodity devices, but rely on strong security assumptions which are hard to achieve in practice [1, 16, 21, 34, 39]. Additionally, they allow a verifier to attest 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 [8, 15, 23, 32]. These architectures have only been implemented as prototypes and their future availability in commodity devices is uncertain.

Contributions. In this work, we present a novel secure code update scheme for wireless mesh networked commodity low-end embedded devices. As opposed to existing hardware-based approaches, we require only minimal assumptions on secure hardware, which makes our scheme applicable to many existing low-end embedded devices. Nevertheless, by relying on lightweight secure hardware, we achieve much stronger security guarantees than existing software- and PoSE-based approaches. This, in particular, allows us to provide secure code updates for groups of devices. Our scheme allows only fresh and authenticated updates to be installed on devices. During a proper code update execution, 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 each others' genuine and up-to-date software state 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 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 are interconnected via ZigBee. Simulation results demonstrate that our scheme scales well and is practically usable in networks with tens of thousands of devices.

Structure. In Section 2, we summarize existing work. Section 3 presents our system model, device requirements, and our adversary model. In Section 4, we show how the device requirements can be implemented on commodity devices. Section 5 describes our secure code update scheme. In Section 6, we evaluate the performance of the proposed scheme. Finally, Section 7 concludes this work.

2 Related Work

Code Updates. The process of updating software or firmware present in embedded devices is referred to as over-the air programming (OTA), firmware over-the-air (FOTA), code update, software update, or firmware update. Common research topics are transmission reliability, transmission scalability, update size minimization, and energy efficiency [13, 18, 27, 36]. Moreover, several papers explicitly focus on security aspects and use digital signatures to ensure code update freshness, authenticity, and integrity [20, 26, 29, 42]. In addition, these works offer features like denial-of-service resilience, extra small or efficient signatures, or support for multiple code update initiators with different privileges. However, conventional code update techniques only perform unidirectional verification. Embedded systems verify the integrity and authenticity of code updates, but the initiator of the code update is unable to verify whether embedded systems indeed install the code update appropriately.

Remote Attestation. Remote attestation is a mechanism that allows a third party to verify the software state of a remote system. Consequently, by performing remote attestation after the execution of a code update, its correct installation can be verified. Software-based attestation mechanisms do not require secure hardware and thus can be applied in commodity low-end embedded systems or legacy systems [10, 25, 30, 38]. However, they rely on various assumptions like exact time measurements, optimal protocol implementation and execution, or the adversary being passive during attestation. Those assumptions are hard to achieve in practice [1]. By contrast, hardware-based attestation mechanisms provide much stronger security guarantees by relying on secure hardware. As standardized and commercial secure hardware components like ARM TrustZone,

TPM, Intel TXT, or Intel SGX are too complex and too expensive to be used in low-end embedded systems, new security architectures, such as SMART [15], SANCUS [32], TrustLite [23], or TyTan [8], 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. In addition, their remote attestation mechanisms only target the attestation of a single device, which is impractical in mesh network scenarios due to a large communication overhead. We are only aware of two approaches that address efficient attestation of multiple embedded devices. SMATT [33] verifies multiple devices at once by comparing their integrity measurements. On the downside, SMATT requires identical devices, relies on special copy-proof memory, and only enables a probabilistic attack detection rate. SEDA [2] is an efficient and scalable attestation scheme for large heterogeneous embedded system networks. Yet, as SEDA relies on secure hardware that is not available in commodity devices, it is not applicable to currently deployed systems. Regarding secure code updates, SEDA provides only a brief protocol extension that leaves several design decisions open (e.g., protection against rollback attacks), and lacks desirable features (e.g., the exclusion of compromised devices from the network).

Secure Code Updates. Work on secure code updates specifically addresses the problem of verifying that a code update has been securely distributed and correctly installed on a remote embedded system. Seshadri et al. [39] applied a software-based approach to ensure an untampered execution of the software update protocol on a single remote device. However, as mentioned in the last paragraph, software-based solutions provide questionable security guarantees due to their strong assumptions [1]. Perito and Tsudik [34] pursued a different approach and introduced the concept of Proofs of Secure Erasure (PoSE) to secure software updates. PoSE allow a device to prove to a remote party that it is free of malicious code by attesting that it has erased all its memory. In a second step, cleaned devices download the software update and send a MAC of the downloaded code to the verifier to prove the storage of the software update. Recently, Karame et al. [21] enhanced this concept by combining PoSE with All or Nothing Transforms to reduce the time and energy overhead. Nevertheless, both software and PoSE-based approaches rely on the strong assumption that a device proving its correct code update installation is only able to communicate with the verifier, and no other party. Thus, both approaches are impractical for updating multiple networked devices, since they can only provide security if the adversary is not physically present and has not gained control of more than one device in the network.

3 System Requirements and Adversary Model

System Model. We consider a mobile wireless mesh network that consists of various interconnected commodity low-end embedded devices. The devices can be of different type and model, having, for instance, varying computational power, storage capacity, or security functionalities. Devices in the network can

move, but the network topology is assumed to remain static during a single run of the secure code update protocol. We assume that all correctly functioning devices are reachable in the network. Unreachable devices are ignored and they are temporarily regarded as compromised, since it is uncertain whether they will ever contribute to the network again. We further assume that each device $\mathcal{D}_i$ gets initialized and deployed by a trusted network operator $\mathcal{O}$ once (see Section 5.1).

After deployment, the goal of $\mathcal{O}$ is to perform a secure and efficient code update for all devices in the network. Devices conducting a secure code update should ensure that only authentic, untampered, and fresh code updates are installed and that the installation establishes software integrity, thereby undoing potential manipulations made by an attacker. Devices that refuse a correct installation should be identified as manipulated and excluded from the network. This prevents compromised devices from eavesdropping, manipulating transmitted data, or communicating with a remote attacker. Finally, $\mathcal{O}$ should get a report listing all devices that are in a trustworthy, i.e., an up-to-date and unmodified, software state. During code update execution, we assume $\mathcal{O}$ to be connected to at least one device in the network.

Hardware Security Requirements. Our secure code update solution requires the following properties from each device $\mathcal{D}_i$:

(1) *Immutable Code:* A static non-volatile write-protected memory region $\mathcal{R}$ which contains code and data;
(2) *Secure Storage:* A device-dependent unique secret $\mathcal{SK}$ that can only be accessed during the execution of code in $\mathcal{R}$;
(3) *Uninterruptible Execution:* Once code in $\mathcal{R}$ gets executed, execution cannot be interrupted until the control flow intentionally leaves $\mathcal{R}$.

In Section 4, we discuss how these properties can be implemented on commodity low-end embedded devices. We will see that many existent devices provide hardware features that allow for the implementation of these requirements.

Adversary Model. We assume that an adversary has full control over the execution state of a compromised device, and can read all readable storage and write to all writable storage. Furthermore, the adversary has complete control over the communication medium, i.e., all messages sent between devices can be eavesdropped and manipulated. In addition, we assume that the adversary can be physically present and introduce additional hardware to the network.

In contrast, we assume that the adversary does not perform physical attacks on the hardware of the embedded devices. In particular, we presume that the adversary cannot bypass any of the hardware protections described above. Moreover, we do not consider Denial of Service (DoS) attacks in the immediate vicinity of the attacker, since there is no defense against a physically present attacker who cuts the wire or jams the wireless communication medium. We would like to point out that the described limitations on the adversary's capabilities are common for hardware-based attestation or code update schemes [2, 8, 15, 23, 32].

4 Requirements on COTS Low-End Embedded Systems

In the following, we demonstrate how each of the stated hardware security requirements (see Section 3) can be implemented on existing commercial off-the-shelf (COTS) low-end embedded devices.

1st Requirement: Immutable Code. Nowadays, it is common for commodity low-end embedded devices to provide protection of Flash memory. On some devices, the Flash memory can be separated into multiple sections which have dedicated lock bits for read protection, write protection, and also interrupt prevention [3]. Most commonly, the Flash memory is divided into one boot loader section (BLS) and one application section. If the device at hand offers this feature, we propose so store the code region $\mathcal{R}$ in the boot loader section and the rest of the program in the application section. Afterwards, we advise to set the lock bits in a way that write access to the boot loader section is denied. This makes $\mathcal{R}$ immutable. Other devices provide a more fine-grained Flash protection, where various memory regions of different sizes can be marked as read-only memory (ROM) or potentially also as execute-only memory (XOM) [40]. ROM can only be read or executed but not modified. XOM provides even stronger protection, since it can exclusively be executed. If the device at hand offers XOM or ROM, we propose to protect $\mathcal{R}$ using the strongest supported memory protection available on the device, i.e., XOM if available and ROM otherwise. 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 [3, 40].

2nd Requirement: Secure Storage. If the particular device offers separable memory (e.g., a BLS) with lock bits, we suggest that $\mathcal{R}$ and the protected secret $\mathcal{SK}$ are stored in an extra section, isolated from the application code. Next, we propose to configure the lock bits in a way that read access to the separated section is denied if it is performed by code stored outside the separated memory region. Thus, $\mathcal{SK}$ can only be read during the execution of $\mathcal{R}$. If the particular device offers XOM, we propose that $\mathcal{SK}$ is stored in XOM using constants that are loaded into the CPU by MOV instructions during execution. Since the content of XOM cannot be read out, $\mathcal{SK}$ only gets revealed during the execution of $\mathcal{R}$. If the particular device only supports ROM, we suggest to store the code region $\mathcal{R}$ in the boot loader and enforce that $\mathcal{R}$ immediately gets executed when the device starts. Consequently, in order to execute $\mathcal{R}$, the device must restart. In addition, we propose to store $\mathcal{SK}$ in a secure key storage whose access can intentionally be denied until the next device restart. In this way, code in $\mathcal{R}$ can read out $\mathcal{SK}$ once during device start and afterwards deny access to $\mathcal{SK}$. As $\mathcal{R}$ is immutable and immediately gets executed when the device starts, an attacker is unable to access $\mathcal{SK}$. 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. [22, 37]. 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 [40]. Once an EEPROM block is hidden, it is not accessible until the next reboot of the device.

3rd Requirement: Uninterruptible Execution. If the device at hand provides separable memory with lock bits, we suggest to set the lock bits of a separated memory section containing $\mathcal{R}$ 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 $\mathcal{R}$. 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 [17]. A further approach is to always let the default interrupt handler clean up sensitive data before control is handed over to the particular user-defined interrupt handler [41]. Both approaches impose no restrictions, since custom interrupts can still be deployed by modifying the user-defined interrupt handlers.

Summary. This section has shown various measures to implement the three required device properties on low-end embedded systems. Because the described measures are frequently available, our scheme is applicable to a wide range of commodity low-end embedded devices. In Appendix A, we provide an overview of popular low-end embedded development devices and show which of the described security mechanisms are available on each device.

5 Secure Code Update Scheme

Our secure code update scheme comprises two phases: an offline phase (see Section 5.1) and an online phase (see Section 5.2). The offline phase is executed once, before the initial deployment of all devices. In the offline 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.

5.1 Offline Phase

For the purpose of authenticating devices and for implementing a challenge-based protocol to attest and verify appropriate update installations, we use public-key cryptography. In the offline phase, $\mathcal{O}$ thus generates a unique identifier i and a unique signature key pair, consisting of a public key $\mathcal{PK}_i$ and a private key $\mathcal{SK}_i$, for each device $\mathcal{D}_i$. $\mathcal{SK}_i$ is stored in a protected storage, which can only be accessed during the execution of code in the static protected memory region $\mathcal{R}$ (see device requirements in Section 3). Furthermore, each device is equipped with

$\mathcal{O}$	Trusted network operator	$\mathcal{SK}_i$	Secret signing key of entity i
$\mathcal{R}$	Static protected code region	$\mathcal{PK}_i$	Public signing key of entity i
$\mathcal{D}_i$	Device with identity i	$\mathcal{DC}_i$	Device certificate of entity i
$\mathcal{SH}_i$	Secret ECDH key of entity i	$\mathcal{SC}_i$	Software certificate of entity i
$\mathcal{PH}_i$	Public ECDH key of entity i	$\mathcal{CU}_c$	Code update for device class c

Table 1. Notation

a device certificate $\mathcal{DC}_i$ and a software certificate $\mathcal{SC}_i$, both signed by $\mathcal{O}$ with $\mathcal{SK}_O$ ($\mathcal{DC}_i.sig, \mathcal{SC}_i.sig$). $\mathcal{DC}_i$ stores the device class c of $\mathcal{D}_i$, the public key $\mathcal{PK}_i$ of $\mathcal{D}_i$, and the identifier i. $\mathcal{SC}_i$ lists all memory regions on $\mathcal{D}_i$ where the code update routine and the firmware is stored. In addition, $\mathcal{SC}_i$ provides hash values over the data of these memory regions ($\mathcal{SC}_i.hash$). Thus, $\mathcal{SC}_i$ can be used to verify the integrity of the installed software on $\mathcal{D}_i$. In order to indicate the freshness of the software, $\mathcal{SC}_i$ also stores a software version number ($\mathcal{SC}_i.ver$). Moreover, each device initially stores the public key of the trusted network operator $\mathcal{PK}_O$ in the write-protected memory region $\mathcal{R}$. $\mathcal{SC}_i$ and $\mathcal{DC}_i$ are stored in a mutable and unprotected memory region.

Additionally, $\mathcal{O}$ equips each device with the functionality 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 $\mathcal{R}$, while the rest of the code (including the actual code update functionality) is stored in a mutable and unprotected memory region (see Figure 1). Table 5.1 summarizes relevant definitions used in the offline and the online phase.

5.2 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 $\mathcal{R}$. The attestation routine verifies the integrity of the installed software and ensures that the device passes execution to an unmodified and up-to-date software. Additionally, the attestation routine generates an attest which proves that the device is in a trustworthy software state by certifying an untampered and complete execution of the attestation routine. In the third stage, neighboring devices exchange and verify each others' software integrity attest. 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 1 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.

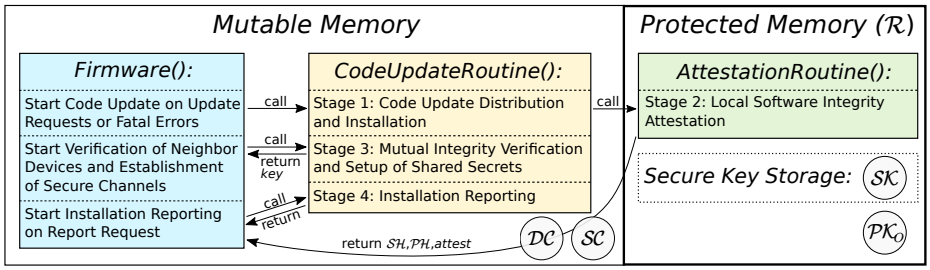

Fig. 1. Illustration of memory layout and control flow of the online phase.

Stage 1: Code Update Distribution and Installation. The online phase starts with the network operator $\mathcal{O}$ 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* must be able to address multiple device classes. For each device class c in the network, *cupkg* contains a software certificate $\mathcal{SC}_c$. $\mathcal{SC}_c$ specifies the correct software configuration for a device of type c, after the installation of the code update. In addition, all contained $\mathcal{SC}$s store the current *cupkg* version number ($\mathcal{SC}.ver = cupkg.no$). Furthermore, for each device class c that should be updated, *cupkg* contains code update data $\mathcal{CU}_c$. $\mathcal{CU}_c$ comprises the binary code of the update and installation instructions (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 1). Next, the code update routine receives *cupkg* and stores it in a free memory region. Devices that received *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 where devices forward the package to neighboring devices that have not yet received *cupkg* (see Figure 2). Since efficient and secure code disseminations in wireless mesh networks are well-understood [13, 18, 20, 26, 27, 29, 36, 42], we will not dwell on the distribution of *cupkg*, but instead assume that eventually each device in the network receives *cupkg*. This also includes scenarios in which *cupkg* is too large to fit into the free memory of devices and must be transmitted in multiple smaller chunks.

Devices that received, verified, and forwarded *cupkg* to their neighbors check whether *cupkg* comprises a new code update for their device class. Thereto, each device $\mathcal{D}_i$ examines whether *cupkg* contains a $\mathcal{CU}_k$ for the local target device class specified in $\mathcal{DC}_i$. If this is the case, a device uses the installation instructions in $\mathcal{CU}_k$ 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

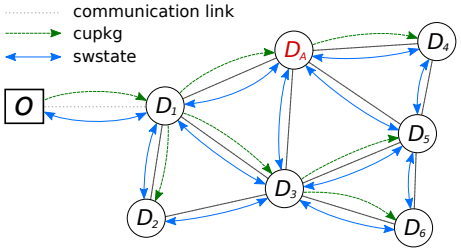

Fig. 2. *Cupkg* distribution in stage 1 and *swstate* message exchange in stage 3.

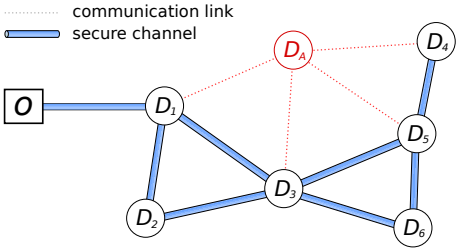

Fig. 3. Establishment of secure channels in stage 3 in presence of an adversary $\mathcal{D}_A$.

local software certificate to the new software certificate for their device class and 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 recover devices whose software accidentally became misconfigured or defective remotely.

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 memory region $\mathcal{R}$ (see Figure 1), this process requires a reboot on certain commodity devices (see Section 4). As illustrated in Algorithm 1, the attestation routine starts with the retrieval of the protected secret signing key $\mathcal{SK}$. Next, the authenticity of the software certificate $\mathcal{SC}$ is ensured by verifying whether $\mathcal{SC}$ was signed by $\mathcal{O}$. If this is the case, $\mathcal{SC}$ 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 $\mathcal{SC}$ are taken and compared to the expected reference values specified in $\mathcal{SC}.hash$. 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 $(\mathcal{SH}, \mathcal{PH})$ [28] and computes *attest* by signing $\mathcal{PH}$ and $\mathcal{SC}.ver$ with $\mathcal{SK}$. Afterwards, it is ensured that no information about the secret signing key $\mathcal{SK}$ gets leaked (denoted by the execution of HideSecret()). As shown in Section 4, 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 $\mathcal{SH}$, $\mathcal{PH}$, and *attest* are passed to the firmware (see Figure 1). The entry point of the firmware is hardcoded in $\mathcal{R}$. 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 buggy *cupkg*.

We would like to point out, that a valid *attest* proves that $\mathcal{D}$ runs a firmware as well as a code update routine whose integrity was successfully verified using a software certificate with the version $\mathcal{SC}.ver$. One reason for this are the three

Algorithm 1. Execution of *AttestationRoutine()* (located in $\mathcal{R}$).

```
1: procedure AttestationRoutine(SC)
2:     SK ← RetrieveSecret()
3:     if Verify(PK_O; SC.sig; SC.content) and CheckCodeIntegrity(SC.hash) then
4:         (SH, PH) ← GenKey()
5:         attest ← Sign(SK; PH||SC.ver)
6:         HideSecret(SK)
7:         StartFirmware(SH, PH, attest)
8:     else
9:         HideSecret(SK)
10:        StartCodeUpdateRoutine()
11: end procedure
```

device properties (see Section 4). They prevent an adversary from tampering with the attestation routine, accessing $\mathcal{SK}$ 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* while not executing the attestation routine from the beginning. This is due to the first instructions of the attestation routine which retrieve $\mathcal{SK}$ 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 $\mathcal{SK}$ gets leaked, that the software integrity of $\mathcal{D}$ conforms to $\mathcal{SC}$, and that the firmware, and no unverified code, gets executed next. Tampering with the input of the attestation routine is also not promising for the adversary. The only mutable data that the attestation routine relies on is $\mathcal{SC}$. However, $\mathcal{SC}$'s integrity is verified before it is used to check the local software integrity. Using old $\mathcal{SC}$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 looks for immediate neighbor devices in the network. 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. Thereto, $\mathcal{D}_i$ generates a $swstate_i$ message comprising $attest_i$, $\mathcal{PH}_i$, and $\mathcal{DC}_i$, 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 2). Next, both devices invoke the execution of stage three in their code update routines (see Figure 1) to verify each others' integrity of the software state and to establish a shared secret. Algorithm 2 illustrates this process in pseudocode.

In order to verify the software state of $\mathcal{D}_n$, $\mathcal{D}_i$ initially checks $\mathcal{DC}_n$ using $\mathcal{PK}_O$. Next, $\mathcal{D}_i$ verifies whether $attest_n$ corresponds to the received $\mathcal{PH}_n$ and the latest software version, which $\mathcal{D}_i$ stores in its local software certificate ($\mathcal{SC}_i.ver$). A successful verification ensures that $\mathcal{D}_n$ is in a software state that corresponds

Algorithm 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     Verify(PK_O; DC_n.sig; DC_n.content)
5:     and    Verify(DC_n.PK_n; attest_n; PH_n||SC_i.ver)
6:     then   key ←KeyExchange(SH_i, PH_n)
7:     return key
8: end procedure
```

to an $\mathcal{SC}$ 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 $\mathcal{PH}_n$. If the verification of $\mathcal{DC}_n$ and $attest_n$ is successful, $\mathcal{D}_i$ uses its own secret ECDH key $\mathcal{SH}_i$ and $\mathcal{D}_n$'s public ECDH key $\mathcal{PH}_n$ 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 likewise 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 1). 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 avoidable 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 $\mathcal{SH}$ 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 3 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 $\mathcal{DC}$ 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 1). The report request is used to construct a spanning tree whose root is $\mathcal{O}$. Thereto, $\mathcal{D}_i$ broadcasts the request over secure channels to all trust-

worthy 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.

Nevertheless, listing the precise identifiers of all devices in the network causes a considerable transmission 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.

6 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 [5]. For the signature scheme, we used an Edwards-curve Digital Signature Algorithm (EdDSA) called Ed25519, which is based on Curve25519 [7]. We implemented the hash function using SHA-512, while the secure and authentic channel uses AES in Galois/Counter Mode (AES-GCM). In Appendix B, we outline measurements of the network performance and the cryptographic implementations.

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 $\mathcal{SC}$ (ca. 212

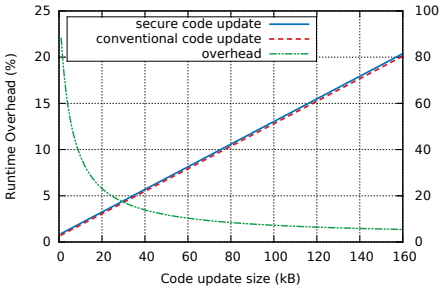

Fig. 4. Single device runtime performance with varying code update sizes.

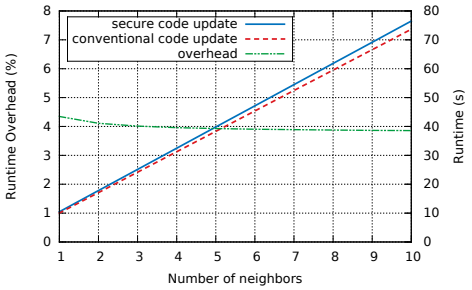

Fig. 5. Single device runtime performance with a varying number of neighbors.

byte), $\mathcal{DC}$ (100 byte), $\mathcal{PK}_O$ (32 byte), $\mathcal{SK}$ (64 byte), ECDH keys (96 byte), and shared secrets (32 byte per neighbor). Hence, with k being the number of neighboring devices, the storage overhead for data adds up to $504 + 32 \cdot k$ bytes.

Another 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 $\mathcal{R}$. 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 [4], we were able to reduce the size of $\mathcal{R}$ to 7.7 kB, albeit increasing the runtime for cryptographic operations.[1] This smaller implementation makes our scheme applicable to all commodity low-end embedded devices listed in Appendix A, since all of them offer at least 8 kB of protectable Flash memory. Reusing the signature scheme in $\mathcal{R}$, further 15.1 kB of code in mutable memory are consumed to implement, among others, the network communication, the key exchange, and the encryption and decryption for the secure channel. 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 respectively 31.6 % and 8.9 % of the totally available storage on the Stellaris platform.

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 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

[1] 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 [12, 31].

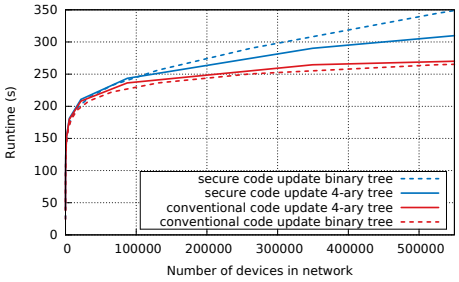

Fig. 6. Network runtime performance with varying network sizes.

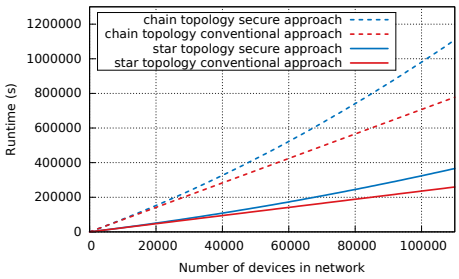

Fig. 7. Network runtime performance with varying network sizes.

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. The figure 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, the runtime overhead decreases from 22.0% down to 1.4% with an increasing size of the code update.

Figure 5 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 each others' software state during protocol execution. Thus, with a varying number of neighbor devices, the runtime overhead remains rather constant at circa 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 6 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 the average size of the report (i.a., 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 7 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 7 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.

7 Conclusion

In this work, we presented a novel secure code update scheme for large mesh networks composed of commodity low-end embedded devices. Our scheme offers desirable security features for the application scenario of patching software vulnerabilities in these systems. Properly executed, our scheme enforces that after code update installation a device runs only 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. We demonstrated that our scheme is applicable to a broad range of popular low-end embedded systems without requiring any hardware modifications. Therefore, our solution can be retrofitted to many currently deployed systems. 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 low-end embedded device networks to increase in size.

References

1. Armknecht, F., Sadeghi, A.R., Schulz, S., Wachsmann, C.: A security framework for the analysis and design of software attestation. In: ACM SIGSAC Conference on Computer & Communications Security (CCS) (2013)
2. Asokan, N., Brasser, F., Ibrahim, A., Sadeghi, A.R., Schunter, M., Tsudik, G., Wachsmann, C.: SEDA: Scalable Embedded Device Attestation. In: ACM SIGSAC Conference on Computer & Communications Security (CCS) (2015)
3. Atmel: Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V Datasheet (2014)
4. Beer, D.: Curve25519 and Ed25519 for low-memory systems (2014), `http://www.dlbeer.co.nz/oss/c25519.html`
5. Bernstein, D.J.: Curve25519: new Diffie-Hellman speed records. In: Public Key Cryptography-PKC 2006. Springer (2006)
6. Bernstein, D.J.: Supercop: System for unified performance evaluation related to cryptographic operations and primitives (2009)
7. Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.Y.: High-speed high-security signatures. Journal of Cryptographic Engineering (2012)
8. Brasser, F., El Mahjoub, B., Sadeghi, A.R., Wachsmann, C., Koeberl, P.: TyTAN: Tiny trust anchor for tiny devices. In: Design Automation Conference (DAC) (2015)
9. Burchfield, T.R., Venkatesan, S., Weiner, D.: Maximizing throughput in zigbee wireless networks through analysis, simulations and implementations. In: Proc. Int. Workshop Localized Algor. Protocols WSNs (2007)
10. Butterworth, J., Kallenberg, C., Kovah, X., Herzog, A.: Bios chronomancy: Fixing the core root of trust for measurement. In: ACM SIGSAC Conference on Computer & Communications Security (CCS) (2013)
11. Costin, A., Zaddach, J., Francillon, A., Balzarotti, D., Antipolis, S.: A large-scale analysis of the security of embedded firmwares. In: USENIX Security (2014)
12. De Clercq, R., Uhsadel, L., Van Herrewege, A., Verbauwhede, I.: Ultra low-power implementation of ECC on the ARM Cortex-M0+. In: Design Automation Conference (DAC) (2014)
13. Dong, W., Chen, C., Bu, J., Liu, W.: Optimizing Relocatable Code for Efficient Software Update in Networked Embedded Systems. ACM Transactions on Sensor Networks (TOSN) (2014)
14. Düll, M., Haase, B., Hinterwälder, G., Hutter, M., Paar, C., Sánchez, A.H., Schwabe, P.: High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. Designs, Codes and Cryptography (2015)
15. Eldefrawy, K., Tsudik, G., Francillon, A., Perito, D.: SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In: NDSS (2012)
16. Francillon, A., Nguyen, Q., Rasmussen, K.B., Tsudik, G.: Systematic Treatment of Remote Attestation. In: IACR Cryptology ePrint Archive (2012)
17. Freescale Semiconductor: Using the Kinetis Flash ExecuteOnly Access Control Feature - 6.3 Entry into execute-only code on the ARM Cortex-M4 core (2015)
18. Hagedorn, A., Starobinski, D., Trachtenberg, A.: Rateless deluge: Over-the-air programming of wireless sensor networks using random linear codes. In: IEEE International Conference on Information Processing in Sensor Networks (2008)
19. Hanna, S., Rolles, R., Molina-Markham, A., Poosankam, P., Fu, K., Song, D.: Take two software updates and see me in the morning: The case for software security evaluations of medical devices. In: Proceedings of the 2nd USENIX Workshop on Health Security and Privacy (HealthSec) (2011)

20. He, D., Chen, C., Chan, S., Bu, J.: SDRP: A secure and distributed reprogramming protocol for wireless sensor networks. IEEE Industrial Electronics (2012)
21. Karame, G.O., Li, W.: Secure Erasure and Code Update in Legacy Sensors. In: Trust and Trustworthy Computing. Springer (2015)
22. Katzenbeisser, S., Kocabaş, Ü., Rožić, V., Sadeghi, A.R., Verbauwhede, I., Wachsmann, C.: PUFs: Myth, fact or busted? A security evaluation of physically unclonable functions (PUFs) cast in silicon. In: Cryptographic Hardware and Embedded Systems–CHES 2012. Springer (2012)
23. Koeberl, P., Schulz, S., Sadeghi, A.R., Varadharajan, V.: TrustLite: a security architecture for tiny embedded devices. In: ACM European Conference on Computer Systems (2014)
24. Kosmal, M.: SharedAES-GCM, `https://github.com/mko-x/SharedAES-GCM`
25. Kovah, X., Kallenberg, C., Weathers, C., Herzog, A., Albin, M., Butterworth, J.: New results for timing-based attestation. In: IEEE Security and Privacy (S&P) (2012)
26. Krontiris, I., Dimitriou, T.: Scatter-secure code authentication for efficient reprogramming in wireless sensor networks. International Journal of Sensor Networks (2011)
27. Kulkarni, S., Wang, L.: Energy-efficient multihop reprogramming for sensor networks. ACM Transactions on Sensor Networks (TOSN) (2009)
28. Law, L., Menezes, A., Qu, M., Solinas, J., Vanstone, S.: An efficient protocol for authenticated key agreement. Designs, Codes and Cryptography (2003)
29. Law, Y.W., Zhang, Y., Jin, J., Palaniswami, M., Havinga, P.: Secure rateless deluge: Pollution-resistant reprogramming and data dissemination for wireless sensor networks. EURASIP Journal on Wireless Communications and Networking (2011)
30. Li, Y., McCune, J.M., Perrig, A.: VIPER: verifying the integrity of PERipherals' firmware. In: ACM SIGSAC Conference on Computer & Communications Security (CCS) (2011)
31. Mackay, K.: Micro-ECC, `http://kmackay.ca/micro-ecc/`
32. Noorman, J., Agten, P., Daniels, W., Strackx, R., Van Herrewege, A., Huygens, C., Preneel, B., Verbauwhede, I., Piessens, F.: Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base. In: USENIX Security (2013)
33. Park, H., Seo, D., Lee, H., Perrig, A.: SMATT: Smart Meter ATTestation Using Multiple Target Selection and Copy-Proof Memory. In: Computer Science and its Applications. Springer (2012)
34. Perito, D., Tsudik, G.: Secure Code Update for Embedded Devices via Proofs of Secure Erasure. In: ESORICS. Springer (2010)
35. Rios, B.: Owning a Building: Exploiting Access Control and Facility Management Systems. Black Hat ASIA (2014)
36. Rossi, M., Bui, N., Zanca, G., Stabellini, L., Crepaldi, R., Zorzi, M.: SYNAPSE++: code dissemination in wireless sensor networks using fountain codes. Mobile Computing, IEEE Transactions on (2010)
37. Schrijen, G.J., van der Leest, V.: Comparative analysis of SRAM memories used as PUF primitives. In: Conference on Design, Automation & Test in Europe (DATE) (2012)
38. Seshadri, A., Luk, M., Perrig, A.: SAKE: Software attestation for key establishment in sensor networks. In: Distributed computing in sensor systems. Springer (2008)
39. Seshadri, A., Luk, M., Perrig, A., van Doorn, L., Khosla, P.: SCUBA: Secure code update by attestation in sensor networks. In: Proceedings of the 5th ACM workshop on Wireless security. ACM (2006)

40. Texas Instruments: Stellaris LM4F120H5QR Microcontroller Data Sheet, 2013
41. Texas Instruments: Software IP Protection on MSP432P4xx Microcontrollers - 10.1 Interrupt Handling in IP Protected Secure Zone (2015)
42. Ugus, O., Westhoff, D., Bohli, J.M.: 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. ACM (2009)

A Commodity Low-End Embedded Device Overview

Device	Immutable Code	Secure Storage	Uninterruptible Execution	Approach
panStamp AVR2	✓	✓	✓	$\mathcal{R}$ in BLS; BLB0=1; BLB1=4; BOOTRST=0
TinyDuino	✓	✓	✓	$\mathcal{R}$ in BLS; BLB0=1; BLB1=4; BOOTRST=0
Arduino UNO	✓	✓	✓	$\mathcal{R}$ in BLS; BLB0=1; BLB1=4; BOOTRST=0
RFduino	✓	✓	✓	Use MPU: Store $\mathcal{R}$ in R0; PALL=0; PR0=0
XinoRF	✓	✓	✓	$\mathcal{R}$ in BLS; BLB0=1; BLB1=4; BOOTRST=0
Ciseco	✓	✓	✓	$\mathcal{R}$ in BLS; BLB0=1; BLB1=4; BOOTRST=0
Pinoccio	✓	✓	✓	$\mathcal{R}$ in BLS; BLB0=1; BLB1=3; BOOTRST=0
Nanode	✓	✓	✓	$\mathcal{R}$ in BLS; BLB0=1; BLB1=4; BOOTRST=0
Arduino Yun	✓	✓	✓	$\mathcal{R}$ in BLS; BLB0=1; BLB1=4; BOOTRST=0
OpenPicus Flyport	✓	–	–	GCP=0; GWRP=0; but not sufficient!
Libelium Wasmote	✓	✓	✓	$\mathcal{R}$ in BLS; BLB0=1; BLB1=4; BOOTRST=0
MICA2	✓	✓	✓	$\mathcal{R}$ in BLS; BLB0=1; BLB1=4; BOOTRST=0
Kinetis KW4*Z	✓	✓	✓	$\mathcal{R}$ in XOM + Interrupt Handler in XOM [17]
Arduino M. 2560	✓	✓	✓	$\mathcal{R}$ in BLS; BLB0=1; BLB1=4; BOOTRST=0
TI Stellaris	✓	✓	✓	(1)$\mathcal{R}$ in XOM+Interrupt Handler in XOM[41] (2)$\mathcal{R}$ in ROM (Bootloader) + EEPROMHIDE (3)$\mathcal{R}$ in ROM (Bootloader) + SRAM PUF

Table 2. Overview of popular low-end embedded devices and their security features.

Table 2 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. We based the list on a collection by Postscapes[2]. In particular, we regarded all their listed low-end embedded systems (first 11 devices) and some devices mentioned in their additional resources. *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

[2] `http://postscapes.com/internet-of-things-hardware`

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 secure code update scheme.

B Additional 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 [9]. In addition, we measured an average one-hop round-trip time (RTT) of 13.5 ms with the smallest message size and 18.5 ms with the biggest message size. 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 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 [6, 24]. Recent works have shown that assembler optimized code for low-end embedded systems can improve the performance of cryptographic operations by orders of magnitudes [12, 14]. 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.

Algorithm	**Function**	**Runtime**	**Function**	**Runtime**
ed25519	genKey()	18 ms	keyExchange()	48 ms
	sign(16 *bytes*)	19 ms	verify(16 *bytes*)	51 ms
	sign(1024 *bytes*)	22 ms	verify(1024 *bytes*)	53 ms
AES-GCM	encrypt(16 *bytes*)	0.1 ms	decrypt(16 *bytes*)	0.1 ms
	encrypt(1024 *bytes*)	1.8 ms	decrypt(1024 *bytes*)	1.8 ms
SHA-512	hash(16 *bytes*)	0.4 ms	hash(20480 *bytes*)	54.7 ms
	hash(1024 *bytes*)	3.1 ms	hash(163840 *bytes*)	435.1 ms

Table 3. Crypto Runtime Performance on the Stellaris.