Boot Attestation: Secure Remote Reporting with Off-The-Shelf IoT Sensors

Steffen Schulz[1(✉)], André Schaller[2], Florian Kohnhäuser[2], and Stefan Katzenbeisser[2]

[1] Intel Labs, 64293 Darmstadt, Germany
steffen.schulz@intel.com
[2] Security Engineering Group, TU Darmstadt, CYSEC, Mornewegstrasse 32, 64293 Darmstadt, Germany
{schaller,kohnhaeuser,katzenbeisser}@seceng.informatik.tu-darmstadt.de

Abstract. A major challenge in computer security is about establishing the trustworthiness of remote platforms. Remote attestation is the most common approach to this challenge. It allows a remote platform to measure and report its system state in a secure way to a third party. Unfortunately, existing attestation solutions either provide low security, as they rely on unrealistic assumptions, or are not applicable to commodity low-cost and resource-constrained devices, as they require custom secure hardware extensions that are difficult to adopt across IoT vendors. In this work, we propose a novel remote attestation scheme, named Boot Attestation, that is particularly optimized for low-cost and resource-constrained embedded devices. In Boot Attestation, software integrity measurements are immediately committed to during boot, thus relaxing the traditional requirement for secure storage and reporting. Our scheme is very light on cryptographic requirements and storage, allowing efficient implementations, even on the most low-end IoT platforms available today. We also describe extensions for more flexible management of ownership and third party (public-key) attestation that may be desired in fully Internet-enabled devices. Our scheme is supported by many existing off-the-shelf devices. To this end, we review the hardware protection capabilities for a number of popular device types and present implementation results for two such commercially available platforms.

1 Introduction

In the Internet-of-Things (IoT) low-cost and resource-constrained devices are becoming the fundamental building blocks for many facets of life. Innovation in this space is not only fueled by making devices ever more powerful, but also by a steady stream of even smaller, cheaper, and less energy-consuming "things' that enable new features and greater automation in home automation, transportation, smart factories and cities.

Unfortunately, the novelty of this space combined with dominating market forces to minimize cost and time-to-market also has a devastating impact on security. While it may be tolerable that deployed firmware is not free of bugs [12]

© Springer International Publishing AG 2017
S.N. Foley et al. (Eds.): ESORICS 2017, Part II, LNCS 10493, pp. 437–455, 2017.
DOI: 10.1007/978-3-319-66399-9_24

and vendors have varying opinions about privacy and access control in this new space [21,35], an arguably critical requirement for survivable IoT infrastructures is the capability to apply security patches and recover from compromises [15,37].

The ability to recognize and establish trust in low-cost devices is becoming relevant even in scenarios where devices are not connected to the Internet or not intended to receive firmware updates at all. For instance, SD-cards and USB sticks that are exchanged with third parties can be infected or replaced by malicious hardware in order to attack the host [29]. Bluetooth devices may offer an even larger attack surface, since typically employed security mechanisms were shown to be insufficient [34,42]. Remote attestation is a key security capability in this context, as it allows a third party to identify a remote device and verify its software integrity.

Existing attestation schemes can be classified as timing-based or hardware-based. Timing-based schemes require no secure hardware and thus are applicable to a broad range of devices [20,23,41]. However, they rely on assumptions like exact time measurements, time-optimal checksum functions, and a passive adversary, which have been proven to be hard to achieve in practice [4,10,22]. In contrast, hardware-based attestation schemes provide much stronger security guarantees by relying on secure hardware components. Recent works specifically target the needs of embedded devices to perform remote attestation with a minimum set of secure hardware requirements [13,14,17,30]. Unfortunately, these hardware security features are currently not available in commodity embedded devices. Another direction of research specifically focuses on a major use case of attestation, the verification of firmware integrity after updates. These works often target device classes that cannot be secured using hardware-based attestation approaches, such as legacy and low-end devices [16,18,32,39]. Yet they only address a subset of attestation usages, suffer from the similar limitations as software-based attestation approaches, or employ costly algorithms that involve a high memory and computational overhead.

Contributions. We present a novel approach to remote attestation which is based on load-time authentication. Our scheme is very efficient and well-suited for resource- constrained, embedded devices (cf. Sect. 2). In more detail:

Boot Attestation Concept: Instead of recording measurements in a secure environment, as in traditional TPM-like attestation, our integrity measurements are immediately authenticated as the platform boots. We will argue in Sect. 3, that for the very simple hardware and firmware configurations found in low-end IoT devices, this construction can meet the key goals of remote attestation which many prior works tried to tackle.

Provisioning and 3^{rd} *Party Verification:* In Sect. 4 we describe two extensions that further increase the practicality and completeness of Boot Attestation. First, a key provisioning extension to take ownership of potentially untrustworthy devices. Second, an extension to enable attestation towards untrustworthy third-party verifiers. The latter is a capability that is missing in prior work, but essential when applying a symmetric attestation scheme in the IoT use case.

Minimal HW/SW Requirements: Our proposed attestation scheme offers a new middle-ground between previously proposed timing-based and hardware-based attestation approaches. Boot Attestation does not depend on timing or other execution-side effects which turned out to be difficult to achieve in practice. As we will discuss in Sect. 5, Boot Attestation side-steps hardware requirements that were deemed essential for remote attestation until now [13]. In contrast to prior work, our approach merely requires memory access control enforcement, secure on-DIE SRAM and debug protection.

Analysis and Implementation: In Sect. 6, we examine hardware protection capabilities for a range of existing Commercial Off-the-Shelf Microcontroller Units (COTS MCUs) and describe how they can be programmed to support Boot Attestation *today*. We then describe two concrete implementations, highlighting the practicality and efficiency of our design.

2 System Model and Goals

In this section, we specify our system model, discuss the adversaries' capabilities and describe the general procedure of remote attestation.

2.1 System Model

We consider a setting with two parties, a verifier $\mathcal{V}$ and a prover $\mathcal{P}$. $\mathcal{V}$ is interested in validating whether $\mathcal{P}$ is in a *known-good* software state, and for this purpose engages in a remote attestation protocol with $\mathcal{P}$.

$\mathcal{P}$ is modeled as a commodity, low-cost IoT device as it may be found in personal gadgets, or smart home and smart factory appliances. In order to minimize manufacturing cost and power consumption, such devices tend to be single-purpose MCUs with often just the minimum memory and compute capabilities required to meet their intended application. Modern MCUs combine CPU, memory, basic peripherals, and selected communication interfaces on a single System on Chip (SoC), as illustrated in Fig. 1. Common on-DIE memory comprises SRAM, flash memory, and EEPROM. Additional peripheral devices and bulk memory are often connected externally.

On the software side, MCUs are often programmed bare-metal, with the SDK building necessary drivers and libraries into a single application binary (firmware image). Some platforms also implement additional stages, e.g. a smart watch loading "companion apps" at runtime. The firmware image is typically initialized by an immutable component, such as a boot ROM or bootloader, which reduces the risk of permanently disabling a device ("bricking"). When programmed via low-level interfaces such as JTAG, many devices also allow to customize this early stage(s) of boot. We will revisit this property when implementing our Root of Trust (RoT) in Sect. 6.

Note that in the IoT context, the attestation verifier $\mathcal{V}$ is typically the owner of $\mathcal{P}$ (e.g., fitness trackers or USB thumb drives) or an operator who is responsible for managing $\mathcal{P}$ on behalf of the owner (e.g., smart factory or smart city).

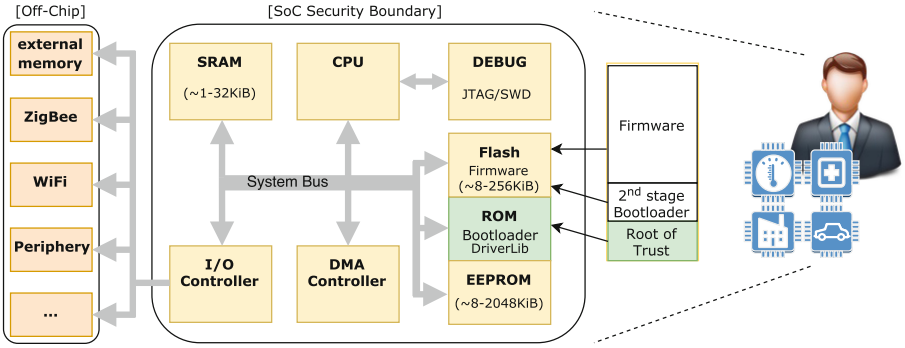

Fig. 1. Typical hardware configuration of a IoT MCU. We consider on-chip components as secure against "simple" hardware attacks (SoC Security Boundary).

2.2 Adversary Model

The adversary $\mathcal{A}$ controls the communication between $\mathcal{V}$ and $\mathcal{P}$ and can compromise the firmware of $\mathcal{P}$ at will. In more detail, $\mathcal{A}$ is granted full control over the communication channel (Dolev-Yao model) and thus can eavesdrop, inject, modify, or delete any messages between $\mathcal{V}$ and $\mathcal{P}$. $\mathcal{A}$ can also compromise the higher-level firmware on $\mathcal{P}$, i.e., the MCU application, whereupon $\mathcal{A}$ has full control over the execution and can read from and write to any memory.

However, we assume that $\mathcal{A}$ is unable to bypass hardware protection mechanisms, such as reading data from memory regions that are explicitly protected by hardware. We also exclude a comprehensive discussion of physical attacks as these require an in-depth analysis of the particular hardware design and are outside the scope of this work. Instead, we consider only a simple hardware adversary who may attempt to access documented interfaces such as JTAG, and replace or manipulate external SoC components like external memory or radios (cf. Fig. 1). We also assume that the verifier does not collaborate with $\mathcal{A}$, in particular, $\mathcal{V}$ will not disclose the attestation key to $\mathcal{A}$. However, this assumption is reduced when introducing third party verifiers in Sect. 4.

2.3 Remote Attestation Game

Remote Attestation is a security scheme where a verifier $\mathcal{V}$ wants to gain assurance that the firmware state of the prover $\mathcal{P}$ has not been subject to compromise by $\mathcal{A}$. Following the common load-time attestation model [31], we define the firmware state of $\mathcal{P}$ as an ordered set of k binary measurements $M = (m_1, m_2, \ldots, m_k)$ that are taken as $\mathcal{P}$ loads its firmware for execution. Since the chain of measurements is started by the platform's Root of Trust (RoT), it is assumed that any modification to the firmware state is reliably reflected in at least one measurement m_x.

To gain assurance on the actual state M' of $\mathcal{P}$, $\mathcal{V}$ and $\mathcal{P}$ engage in a challenge-response protocol. This culminates in the construction of an attestation report

$r \leftarrow attest_{\mathsf{AK}}(c, M')$ at $\mathcal{P}$, where c is a random challenge and AK is an attestation key agreed between $\mathcal{P}$ and $\mathcal{V}$. $\mathcal{V}$ accepts $\mathcal{P}$ as trustworthy, i.e., not compromised, if the response r is valid under chosen values (c, AK) and an expected known-good state M (i.e., $M' = M$).

3 Boot Attestation

In this section, we introduce our Boot Attestation concept and protocol, extract hardware requirements and analyze its security with regard to Sect. 2.3.

3.1 Implicit Chain of Trust

Traditional attestation schemes collect measurements in a secure environment, such as a TPM or TEE, which can be queried at a later time to produce an attestation report. They support complex software stacks comprising a large set of measurements and allow a multitude of verifiers to request subsets of these measurements, depending on privacy and validation requirements.

In contrast, our approach is to authenticate measurements m_x of the next firmware stage x immediately into an authenticated state M_x, before handing control to the next firmware stage. This way, m_x is protected from manipulations by any subsequently loaded application firmware. The new state M_x is generated pseudo-randomly and the previous state M_{x-1} is discarded. This prevents an adversary from reconstructing prior or alternative measurement states. The final state M_k, seen by the application, comprises an authenticated representation of the complete measurement chain for reporting to $\mathcal{V}$:

$$M_x \leftarrow PRF_{\mathsf{AK}}(M_{x-1}, m_x)$$

As typically no secure hardware is available to protect AK in this usage, we generate pseudo-random sub-keys AK_x and again discard prior keys AK_{x-1} before initiating stage x:

$$\mathsf{AK}_x \leftarrow KDF_{\mathsf{AK}_{x-1}}(m_x), \text{ with } \mathsf{AK}_0 \leftarrow \mathsf{AK}$$

Note that we can instantiate PRF and KDF using a single HMAC. The measurement state M_x has become implicit in AK_x and does not have to be recorded separately.

The approach is limited in the sense that the boot flow at $\mathcal{P}$ dictates the accumulation of measurements in one or more implicit trust chains M. However, for the small, single-purpose IoT platforms we target here, there is typically no need to attest subsets of the firmware state as it is possible with TPM PCRs. The next section expands this idea into a full remote attestation protocol.

3.2 Remote Attestation Protocol

Figure 2 provides an overview of a possible remote attestation protocol utilizing the implicit chain of trust and a symmetric shared attestation key AK. On the right-hand side, the prover $\mathcal{P}$ builds its chain of trust from the Root of Trust to a possible stage 1 (bootloader) and stage 2 (application). Once booted, the prover may be challenged by $\mathcal{V}$ to report its firmware state by demonstrating possession of the implicitly authenticated measurement state AK_2.

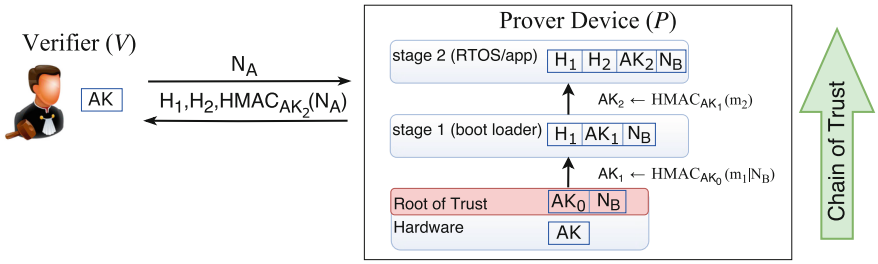

Fig. 2. Schematic overview of one possible instantiation of our Boot Attestation scheme as part of a remote attestation protocol.

The detailed protocol works as follows. The prover hardware starts execution at the platform Root of Trust (RoT). This "stage 0" has exclusive access to the root attestation key $\mathsf{AK}_0 \leftarrow \mathsf{AK}$ and an optional boot nonce N_B. It derives AK_1 as $HMAC_{\mathsf{AK}_0}(N_B, m_1)$, with $m_1 := (start_1, size_1, H_1)$ defined as the binary measurement of the firmware stage 1. Before launching stage 1, the RoT must purge intermediate secrets from memory and lock AK against further access by application firmware. Execution then continues at stage 1 using the intermediate attestation key AK_1 and measurement log (H_1, N_B)[1].

The scheme continues through other boot stages $x \in \{1, \ldots, k\}$ until the main application/runtime has launched in stage k. In each stage, a measurement m_{x+1} of the next firmware stage is taken and extended into the measurement state as $\mathsf{AK}_{x+1} \leftarrow HMAC_{\mathsf{AK}_x}(m_{x+1})$. The prior attestation key AK_x and intermediate values of the $HMAC()$ operation are purged from memory so that they cannot be accessed by stage $x+1$. Finally, the measurement log is extended to aid the later reconstruction and verification of the firmware state at $\mathcal{V}$.

Once $\mathcal{P}$ has launched the final stage k, it may accept challenges $c \leftarrow N_A$ by a remote verifier to attest its firmware state. For $\mathcal{P}$, this simply involves computing a proof of knowledge $r \leftarrow HMAC_{\mathsf{AK}_k}(N_A)$ and sending it to $\mathcal{V}$ together with the measurement log. Using this response, the verifier $\mathcal{V}$ can reconstruct the state $M' = (m'_1, \ldots, m'_k)$ claimed by $\mathcal{P}$ and the associated AK_k. $\mathcal{V}$ can then validate and accept $\mathcal{P}$ if $M' = M$ and $r = HMAC_{\mathsf{AK}_k}(N_A)$.

[1] We consider $(start_x, size_x)$ as well-known parameters here, since the individual m_x would typically encompass the complete firmware image at a particular stage.

Note that for the devices we target, k tends to be very low. Typical MCUs load only one or two stages of firmware, which helps keeping the validation effort at $\mathcal{V}$ manageable even for large amounts of valid platforms (AK, M).

We emphasize that the protocol described here only considers the core attestation scheme. A complete solution should also consider authorizing $\mathcal{V}$ towards $\mathcal{P}$, protecting the confidentiality of the attestation report and linking the attestation to a session or otherwise exchanged data. As part of an authorized attestation challenge c', $\mathcal{V}$ may also include a command to update N_B and reboot $\mathcal{P}$ to refresh all AK_x. However, while the implications of managing N_B are discussed in Sect. 3.3, the detailed choices and goals are application-dependent and outside the scope of this work.

3.3 Security Analysis

In the following, we analyze the security of Boot Attestation based on the adversary model and attestation game defined in Sects. 2.2 and 2.3. We will show that Boot Attestation is able to provide the same security as all load-time attestation approaches, such as TPM-based attestation schemes [28]. For this purpose, we consider the relevant attack surface in terms of *network*, *physical/side-channel* as well as *load-time* and *runtime* compromise attacks.

Network Attacks. The adversary $\mathcal{A}$ can eavesdrop, synthesize, manipulate, and drop any network data. However, the employed challenge-response protocol using the shared secret AK trivially mitigates these attacks. More specifically, any manipulation of assets exposed on the network, including H_1, H_2, N_A or the attestation response r, is detected by $\mathcal{V}$ when reconstructing AK_k and validating $r = HMAC_{\mathsf{AK}_k}(N_A)$. The attestation nonce N_A mitigates network replay attacks. $\mathcal{A}$ can cause a DoS by dropping messages, but $\mathcal{V}$ will still not accept $\mathcal{P}$.

Since AK is a device-specific secret, the intermediate keys AK_x and final response r are uniquely bound to each individual device. This allows Boot Attestation to function seamlessly with emerging swarm- attestation schemes, where the same nonce N_A is used to attest many devices at once [1,6,9].

Physical and Side-Channel Attacks. $\mathcal{A}$ may attempt simple hardware attacks, using SoC-external interfaces to gather information on intermediate attestation keys AK_x or manipulate SoC- external memory and I/O. Boot Attestation assumes basic hardware mechanisms to protect debug ports and protect intermediate values in memory (cf. Sect. 5). Otherwise, the resilience against hardware attacks heavily depends on the SoC implementation and is outside our scope.

$\mathcal{A}$ could also attempt software side-channel attacks, such as cache, data remanence, or timing attacks. However, as each stage i clears any data besides $\mathsf{N}, H_{i+1}, \mathsf{AK}_{i+1}$, there is no confidential data that a malware could extract from cache, RAM, or flash. Furthermore, the risk of timing side-channels is drastically reduced as root keys are only used initially by the RoT. Implementing a constant-time HMAC operation in the typically used non- paged, tightly coupled SRAM is straightforward.

Load-Time Compromise. $\mathcal{A}$ may compromise the firmware stage a of $\mathcal{P}$ before it is loaded and hence, measured. In this case, $\mathcal{A}$ can access all intermediate measurements $(m_1, \ldots, m_k)$, the nonces (N_B, N_A), and any subsequent attestation keys $(\mathsf{AK}_a, \ldots, \mathsf{AK}_k)$. Note that compromising the RoT (i.e., the initial stage) is outside the capabilities of $\mathcal{A}$. This is a reasonable assumption due to RoT's hardware protection (cf. Sect. 5) and miniscule code complexity (cf. Table 2).

Compromising the intermediate measurement state and keys allows $\mathcal{A}$ building alternative measurement states M'_{a+n} and associated attestation keys AK'_{a+n} for positive integers n. However, $\mathcal{A}$ cannot recover the attestation keys of prior stages $a - n$, as they have been wiped from memory prior to invoking stage a. In particular, $\mathcal{A}$ cannot access the root attestation key AK, which can only be accessed by the RoT. As a result, $\mathcal{A}$ can only construct attestation responses that *extend* on the measurement state M'_a and the associated attestation key AK_a. Moreover, load-time attestation assumes that the measurement chain is appropriately setup to record the compromise, so that (M'_a, AK'_a) already reflect the compromise and cannot be expanded to spoof a valid firmware state M_k.

In practice, successfully recording M'_a will typically require a persistent manipulation or explicit code loading action by the adversary. However, this is a well-known limitation of load-time attestation and also affects the TPM and other load-time attestation schemes.

Following a firmware patch to return stage a into a well-known, trustworthy component, a new measurement and associated key chain is produced starting at stage a. $\mathcal{A}$ is unable to forsee this renewed key chain, as this would require access to at least AK_{a-1}.

Runtime Compromise. $\mathcal{A}$ may also compromise the firmware stage a at runtime, after is measured, e.g., by exploiting a vulnerability that leads to arbitrary code execution. In this case, $\mathcal{A}$ would have access to the correct (unmodified) attestation key AK_a, could bypass the chain of trust, and thus win the attestation game. Note that this is a generic attack affecting all load-time attestation schemes, including the TPM [28]. Even platforms supporting a Dynamic Root of Trust for Measurement (DRTM) cannot detect runtime attacks after the measurement was performed. However, Boot Attestation performs slightly worse in this case since $\mathcal{A}$ may additionally record AK_a and replay it later on to simulate alternative measurement states and win the attestation game, even after reboot. Nevertheless, Boot Attestation allows recovering the platform and returning into a trustworthy state, in the same way as with other load-time attestation schemes, by patching the vulnerable firmware stage a and performing a reboot. This leads to a refresh of attestation keys $\mathsf{AK}_a, \ldots, \mathsf{AK}_k$ in a way that is unpredictable to $\mathcal{A}$, thus enabling $\mathcal{V}$ to attest the proper recovery of $\mathcal{P}$ and possibly reprovision application secrets.

To further mitigate the risk of a compromised AK_a, $\mathcal{V}$ may also manage an additional boot nonce N_B as introduced in Sect. 3.2. Depending on the particular usage and implementation of $\mathcal{P}$, N_B could be incremented to cause a refresh of the measurement chain without provisioning new firmware. For instance, MCUs

in an automotive on-board network may regularly receive new boot nonces for use on next boot/validation cycle.

4 Extensions for Real-World Use

In the following, we discuss extensions to our attestation scheme that are commonly not considered in prior work, but which are fundamental for real-world application in IoT. The first extension provides support for provisioning an attestation key and requires a slight extension of our HW requirements. The second extension is a software-only solution to support verification of attestation reports by untrusted third parties.

4.1 Attestation Key Provisioning

In many cases, it is desirable to provision a new root attestation key AK to a possibly compromised platform. Examples include a user taking ownership of a new or second-hand device, or issuing a fresh AK after compromise of the verifier. To realize this, we follow the TCG concept of provisioning a device-unique *endorsement key* EK as part of manufacturing of $\mathcal{P}$. EK is a symmetric key shared between $\mathcal{P}$ and the manufacturer $\mathcal{M}$. This allows $\mathcal{M}$ to authorize AK key provisioning requests from $\mathcal{V}$ to $\mathcal{P}$, while at the same time ensuring the authenticity of the provisioning target $\mathcal{P}$ to $\mathcal{V}$.

Provisioning AK based on EK can be realized with a number of key exchange or key transport protocols implemented in RoT. We omit a detailed instantiation here due to the lack of space. However, we like to call out the slightly extended key protection requirement for supporting such a scheme. In particular, AK provisioning requires the AK storage to be writable by RoT but read/write-locked for subsequent FW stages (cf. Sect. 5).

4.2 Third-Party Verification

In many IoT usages, MCUs operate not just with a single trusted device owner or manager, but can establish a variety of interactions with user platforms, infrastructure components and cloud backends.

However, as the final attestation key AK_k is a critical asset during attestation, sharing it with all possible verifiers would significantly reduce the confidence into the scheme. To tackle this issue, which is shared by all existing symmetric attestation schemes [13,30], we extend Boot Attestation to allow for potentially untrusted third-party verifiers.

For this purpose, we turn the original verifier $\mathcal{V}$ into a Certification Authority ($\mathcal{CA}$). $\mathcal{CA}$ and $\mathcal{P}$ do not use AK_k directly, but instead generate a key pair based on the pseudo-random input AK_k. In order to attest $\mathcal{P}$ by third-party verifies $\mathcal{V}'$, only the public key computed from AK_k must be distributed.

In practice, one would store AK in a secure environment at the owner or manufactuer and only distribute and use valid public keys based on expected firmware measurements. The detailed protocol works as follows:

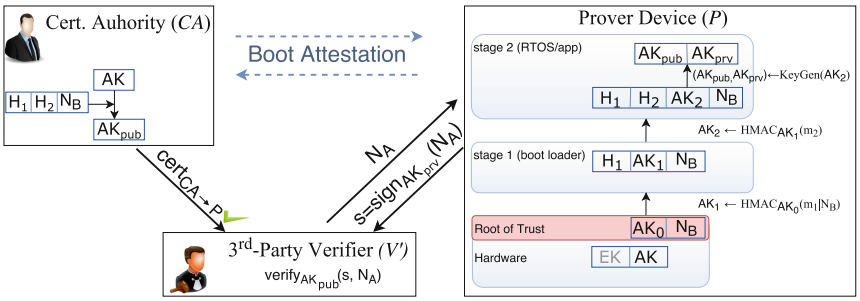

Fig. 3. Third-party verification using a trusted $\mathcal{CA}$. The optional boot attestation phase is depicted with dashed arrows.

Initially, $\mathcal{P}$ and $\mathcal{CA}$ share a common secret AK and that $\mathcal{P}$ was initialized according to Sect. 3.1, i.e. has derived the correct key AK_k.

This time, $\mathcal{P}$ uses AK_k as pseudo-random input to generate a key pair $(\mathsf{AK}_{prv}, \mathsf{AK}_{pub}) \leftarrow KeyGen(\mathsf{AK}_k)$. This can be done deterministically for example using ECC key generation. Subsequently, $\mathcal{CA}$ receives $(H_1, \ldots, H_k, \mathsf{N}, r = HMAC_{\mathsf{AK}_k}(c))$ from $\mathcal{P}$. Using the intermediate hashes $(H_1, \ldots, H_k)$ and AK, $\mathcal{CA}$ can reproduce AK_k and $\mathcal{P}$'s public key AK_{pub} and publish a certificate $cert_{\mathcal{CA}\rightarrow\mathcal{P}} \leftarrow Sign_{\mathcal{CA}_{prv}}(\mathsf{AK}_{pub})$.

The third party $\mathcal{V}'$ initiates attestation by querying $\mathcal{CA}$ for $\mathcal{P}$'s signed public key $cert_{\mathcal{CA}\rightarrow\mathcal{P}}$. Subsequently $\mathcal{V}'$ challenges the (valid) prover $\mathcal{P}$ for a fresh signature, using a nonce N_A. In turn, $\mathcal{P}$ creates a signature s of N_A $s \leftarrow Sign_{\mathsf{AK}_{prv}}(\mathsf{N}_A)$ and sends s to $\mathcal{V}'$. The third party is now able to infer statements about $\mathcal{P}$'s identity and firmware state. At the same time AK_k is kept secret from $\mathcal{V}'$. An overview of the scheme is shown in Fig. 3.

5 Hardware Requirements

In this section, we describe the hardware requirements of our Boot Attestation scheme in detail. We formulate these as results here and not as system assumptions in Sect. 2.1, since the exploration of alternative remote attestation schemes with minimal hardware requirements has been a major research challenge in recent years [13,14,17,30]. In particular, remote attestation schemes proposed so far still require a secure co-processor or custom hardware security extensions in order to support the secure recording and signing of measurements. Alternative approaches using a software-only root of trust still require strong assumptions on the operating environment and implementation correctness, which has precluded them as a generic attestation solution for IoT [10,22].

Leveraging the implicit chain of trust, our Boot Attestation scheme avoids the requirement for a hardware-isolated attestation runtime. Specifically, we only require the following hardware security features:

[I] RoT Integrity: The RoT is critical to initializing the chain of trust and protecting fundamental platform assets such as AK. Our scheme requires RoT integrity in the sense that it must be impossible for the adversary $\mathcal{A}$ to manipulate the RoT firmware, and that the RoT must be reliably and consistently executed at platform reset. In practice, this requires hardware access control on the RoT code and data region, but also hardware logic to consistently reset the SoC's caches, DMA engines and other interrupt-capable devices in order to reliably execute RoT on power-cycle, soft-reset, deep sleep, and similar events. While RoT integrity is well-understood in terms of supporting secure boot or firmware management, we know of no COTS MCU which natively supports a RoT for attestation. To realize Boot Attestation on COTS MCUs we therefore require an extension of the RoT integrity requirement: The device owner must be able to customize or extend the initial boot phase to implement an attestation RoT, and then lock or otherwise protect it from any further manipulation. As we will show, many COTS MCUs actually offer this level of customization prior to enabling production use.

[II] AK Protection: Boot attestation requires that the root attestation key AK is read-/write-locked ahead of execution of the firmware application. This typically requires the RoT to initialize some form of memory access control and then lock it down, such that it cannot be disabled by subsequent firmware stages. While such lock-able key storage is not a standard feature, we found that most COTS MCUs offer some kind of memory locking or hiding that can be used to meet this requirement (cf. Sect. 3.3).

[II*] AK Provisioning: Provisioning of a new attestation key AK_{new} involves replacement of its previous instance, conducted by the RoT (cf. Sect. 4.1). Hence, in order to support provisioning, AK must further be writable by the RoT exclusively. However, this procedure is preceded by the secure negotiation of AK_{new}. During this process the endorsement key EK is used to provide authorization and confidentiality of the new attestation key AK_{new}. Thus, during key provisioning the RoT must read EK and then lock it against read attempts by latter firmware stages, basically resembling requirement **[II]**.

[III] State Protection: When calculating measurements m_x and attestation keys AK_x, the respective firmware stage must be able to operate in a secure memory that cannot be accessed by later firmware stages or other unauthorized platform components. This includes protecting intermediate values of the HMAC calculation as well as the stack. In practice, this requirement breaks down to operating in memory that is shielded against simple hardware attacks (cf. Sect. 2.2), such as the SoC on-DIE SRAM, and clearing sensitive intermediate values from memory before handing control to the respective next stage.

[IV] Debug Protection: Once programmed and provisioned, the device should reject unauthorized access via external interfaces such as UART consoles, JTAG or SWD debug interfaces [11]. Strictly speaking this requirement is sufficiently addressed if the above integrity and confidentiality requirements are met. However, we list it here as separate requirement since debug access and re-program-

ming protections are typically implemented and enabled separately from the above general implementation requirements.

Overall, we can see that Boot Attestation side-steps requirements for protecting the initial call into the secure environment and inhibiting interrupts during execution - including resets - which are not achievable with established hardware protection mechanisms and therefore also not feasible on commodity COTS MCUs [13,14].

6 Proof of Concept Implementation

We reviewed specifications for a range of popular COTS MCUs with regard to meeting the hardware requirements of Boot Attestation (cf. Sect. 5), including support for AK Provisioning (req. [**II**] *).

All of the platforms we investigated support executing firmware completely within the confines of the SoC, ensuring confidentiality and integrity against external HW manipulation (req. [**III**]). Most of the chips also offer one or more lock bits to disable debug access for production use ([**IV**]). Differences could be observed mainly in the memory access control facilities, with a large variety in the number, granularity and permissions available per physical memory block. In contrast, all of the investigated devices support customization and subsequent locking of the boot "ROM", allowing developers to customize and then integrity-protect the platform Root of Trust in one way or another (req. [**I**]).

An overview of the results is provided in Table 1. Apart from [**I**] RoT Integrity and [**IV**] Debug Protection, we also list the respective options for protecting AK and EK in the AK Provisioning scenario (req. [**II**] *)[2]. As can be seen, Boot Attestation is potentially supported by a wide range of devices. Naturally, a full implementation and validation is required to ensure the respective platform controls are accurately documented and sufficient in practice.

We selected two rather different device types, the Stellaris LM4F120 and the Quark D2000, to evaluate different implementation options and provide an overview of the associated costs. In both cases, the implementation of our scheme comprised extending the RoT for measuring the FW application image and deriving an attestation key, as well as initializing the hardware key protection for AK and EK. Note also that there is no intermediate bootloader stage on these devices as the application image is directly executed by RoT. An overview of the implementation footprint is provided in Table 2.

6.1 Prototype I: TI Stellaris LaunchPad

The TI Stellaris Launchpad [44] implements an ARM Cortex-M4F operating at 80 MHz[3], 32 kB SRAM, 32 kB flash memory and 2 kB EEPROM. The platform is typically used in industrial automation, point of sale terminals and network

[2] If no AK provisioning is desired, EK protection is sufficient to store AK_0 (req. [**II**]).

[3] In our experiments we set the operating frequency to 50 MHz to allow for better comparison with the Intel MCU.

Table 1. List of COTS MCUs and how they meet our hardware requirements.

Device type	CPU	SRAM (kB)/ Flash (kB)/ EEPROM (kB)	RoT Integrity	AK protection	EK protection	Debug protection
ATmega328P	AVR	2/32/1024	Flash	Flash	Flash/PUF	✓
PIC16F1825	PIC16	1/8/256	Flash	Flash/ EEPROM	Flash/ EEPROM	✓
LPC1200	Cortex-M0	4–8/32–128/–	Flash	✗	✗	✓
STM32F100Rx	Cortex-M3	8/64–128/–	Flash	✗	PUF	✗
Stellaris LM4F120	Cortex-M4F	32/256/2048	Flash	Flash	Flash/ EEPROM	✓
Quark MCU D2000	Quark D2000	8/44/–	Flash	Flash (main)	Flash (OTP)	✓
Arduino/ Genuino101	Quark SE C1000	80/392/–	Flash	Flash	Flash	✓

appliances. We use FreeRTOS [33] as a firmware stack, as it is freely available, pre-configured for the Stellaris and as it exhibits a small memory footprint.

Integrity Protected RoT. The Stellaris supports RoT code integrity by enabling execute-only protection to those flash blocks that store the boot loader. In particular, by setting register values of `FMPPEn` and `FMPREn` to '0', read and write access to the bootloader section is disabled while keeping it executable.

Protection of AK and EK. Although the Stellaris provides memory protection for flash [5], we decide not to use it for secure key storage. Despite the fact that individual blocks of flash memory can be read-protected, it is yet possible to execute said blocks. This could allow an attacker $\mathcal{A}$ to extract bits of AK or EK. $\mathcal{A}$ can try to execute respective memory regions and infer information by interpreting resulting execution errors. Instead, we securely store AK and EK on the internal EEPROM module. The Stellaris platform provides register `EEHIDE` that allows for hiding individual 32 B EEPROM blocks until subsequent reset.

PUF-Based Storage of EK. It is also possible to securely store EK using a fraction of the on-chip SRAM as a PUF. Previous work supports the use of SRAM as PUFs for key storage [27,38]. Indeed, the SRAM-based PUF instance of the Stellaris has already been characterized in [19]. Using PUFs as a key storage significantly increases the level of protection, as PUF-based keys are existent only for a limited period [3]. Especially for long-term keys, such as EK, this is a desirable property, which is otherwise hard to achieve on low-cost devices. To evaluate this option, we implemented a Fuzzy Extractor construction based on [7]. On start-up of the device, a fraction of the SRAM start-up values are used as a (noisy) PUF measurement X. Using X and public Helper Data W that was created during a prior enrollment phase, the Fuzzy Extractor can reconstruct EK. For details on the the interaction with SRAM-based PUFs, we refer to [36]. Assuming a conservative noise level of 15 % in the PUF measurements X, which is a common value used in literature [7], and applying a $(15, 1, 15)$ repetition code as part of the Fuzzy Extractor, we achieve an error probability of 10^{-9}.

Debug Protection. The bootloader is further protected from attempts to replace it by malicious code by disabling access to JTAG pins. For this purpose bits `DBG0`, `DBG1` and `NW`, part of register `BOOTCFG` are set to '0'. This leaves a subset of standard IEEE instructions intact (such as boundary scan operations), but blocks any access to the processor and peripherals.

6.2 Prototype II: Intel Quark D2000

The Intel Quark Microcontroller D2000 employs an x86 Quark CPU operating at 32 MHz, 8 kB SRAM, 32 kB main flash memory, as well as two regions (4 kB and 4 kB) of One-Time-Programmable (OTP) flash memory. The Intel D2000 is tailored towards IoT scenarios, where low energy consumption is required. We use the Intel Quark Microcontroller Software Interface (QMSI) [25] and Zephyr RTOS [26] as the standard firmware stack.

Integrity Protected RoT and Debug Protection. 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 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'.

Protection of AK and EK. We store AK in main flash to support updates via key provisioning. One of the D2000 flash protection regions (FPR) is setup and locked by the RoT 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 OTP memory and set the read-lock just at the very end of RoT execution. The read-lock for the lower and upper OTP region is activated by programming bits `ROM_RD_DIS_L` and `ROM_RD_DIS_U` of the `CTRL` register.

6.3 Performance Evaluation

In the following, we present evaluation results for both device types, with focus on memory footprint and runtime. Numbers are given for the RoT and key protection logic. Values for the RoT are further separated with respect to RoT base logic (memory management, setup of data structures) and the HMAC implementation. Runtime results of the HMAC functionality are given for a memory range of 256 bit, i.e., a single HMAC data block, and a 32 kB region that reflects larger firmware measurements. For both, memory footprint and runtime, we further provide numbers with respect to two different compile time optimizations. The detailed results are given in Table 2.

Memory. For memory consumption we consider static code segments (`.text`) and read-only data (`.rodata`) segments of the firmware image. Table 2 lists results for compile optimizations towards size (`-Os`) and runtime (`-O1`). Using

Table 2. Implementation overhead with respect to runtime in milliseconds (left) and memory overhead in Bytes (right) for the TI Stellaris (ARM) and the Intel D2000 (x86), with optimizations for size (`-Os`) and runtime (`-O1`).

	Size (Bytes)				Runtime (ms)			
	ARM		x86		ARM		x86	
Component	`-Os`	`-O1`	`-Os`	`-O1`	`-Os`	`-O1`	`-Os`	`-O1`
Base ROM	702	712	1955	2115	0.79	0.63	6.11	5.93
Root of Trust (RoT)								
Base Logic	336	340	168	193	<0.01	<0.01	<0.01	<0.01
HMAC-SHA2 (256 bit)	1828	1836	1819	2061	3.04	3.04	1.54	1.44
HMAC-SHA2 (32 kB)	1828	1836	1819	2061	312.26	312.26	148.23	145.37
AK Protection								
Flash	—	—	295	337	—	—	0.02	0.02
EEPROM	516	580	—	—	0.01	<0.01	—	—
EK Protection								
Flash	—	—	378	448	—	—	<0.01	0.002
EEPROM	516	580	—	—	0.01	<0.01	—	—
SRAM PUF	1662	1980	—	—	46.44	46.42	—	—

the most memory-efficient setting, the scheme requires a total of ≈3.1 kB on the Stellaris. This may seem large compared to the 700 B footprint of the base ROM image (i.e., excluding the application), but is only 1.22 % of the total available flash. On Intel D2000, our RoT extension consumes 2.6 kB on top of the QMSI stock ROM of 2 kB. This fits well within the total 8 KB available for boot loader code. The application flash is left for use by applications, except for the small part reserved for AK storage.

Runtime. Additional runtime introduced by our scheme mainly results from HMAC operations in order to compute attestation measurements, with the key protection logic introducing only little overhead. The right hand side of Table 2 lists runtime overhead of our implementation. As to be expected, the main overhead is caused by the HMAC function which depends on the concrete size of the next stage to be measured. We give 256 B and 32 kB as reference points to estimate the cost hashing the KDF output and a larger firmware, respectively. The D2000 is more than two times faster in computing authenticated measurements over various memory regions, which is much likely due to faster flash access. In particular, the D2000 requires only 145 ms for hashing 32 kB, whereas the Stellaris takes 312 ms. In contrast, the key protection logic adds negligible runtime for both device types. It takes less than 0.02 ms on the Stellaris and 0.04 ms on the Intel D2000, in the worst-case. Lastly, the SRAM PUF is by far the slowest key storage solution for EK on the Stellaris, taking almost half a second. This is due to costly error correction of the PUF measurements. As a reference, the

unmodified base ROM, without our extension, takes on average 0.7 ms on the Stellaris and 6 ms on the Intel D2000.

7 Related Work

Previous work on attestation addresses *hardware-based* or *timing-based* attestation, *scalable attestation* for groups of devices, and *secure code updates*. verifier, to check the integrity of the software on a remote device, named prover. Prior work is related to *timing-based* or *hardware- based* attestation, *scalable attestation* for groups of devices, or *secure code updates*.

Hardware-based attestation schemes rely on secure hardware, such as Intel SGX or a Trusted Platform Module (TPM) [2,28], that is installed on the prover. Since such secure hardware is typically too expensive and complex to be integrated in low-cost embedded devices, recent works focused on the advancement of new minimalist security architectures [13,14,17,30] which enable hardware-based remote attestation capabilities for small embedded devices. However, these lightweight architectures have not yet reached the market, and hence are not available in commodity low-end embedded devices. Furthermore, even when they are available, there is still the need to secure old systems.

By contrast, timing-based attestation schemes do not require secure hardware and thus are applicable to legacy systems [20,24,40,41]. However, they rely on assumptions that have proven to be hard to achieve in prac tice [4,10,22]. Such assumptions include an optimal implementation and execution of the protocol, exact time measurements, and an adversary who is passive during attestation.

Recent works address a scalable attestation of groups of devices (i.e., device swarms) that are interconnected in large mesh networks [1,6,9]. The basic idea is that neighboring devices mutually attest each other in order to distribute the attestation burden across the entire network. Since these works rely on hardware-based attestation schemes, such as [8,13,17], they could leverage our Boot Attestation scheme to be applicable to a broader range of embedded devices.

The field of secure code updates specifically addresses the challenge of verifying the integrity firmware after ist has been updated. Initial approaches employ software-based attestation techniques [39], and hence inherit their characteristics, mentioned above. Later on, the notion of Proofs of Secure Erasure (PoSE) was introduced to secure code updates [16,32]. PoSE-based approaches build on a challenge-response protocol that requires the prover to fill its entire memory with data, in turn overwriting any malicious code. Although such solutions can be applied to many devices, as they require a small amount of read-only memory, they assume an network adversary to only communicate with the verifier but not the prover device, which is a strong limitation. Recent work focuses on scalable updates in large mesh networks [18]. In contrast to our work, it imposes the use of asymmetric cryptography, involving heavy computational overhead and a large memory footprint. There are also platform-specific security extensions such as cryptoBSL [45] and STM32 PCROP [43]. While they are focused on secure boot and IP protection, it would be interesting to evaluate their use in context of remote attestation and recovery.

8 Conclusions and Future Work

In this work, we explored a novel lightweight remote attestation scheme for low-cost COTS MCUs. We showed that it is possible to narrow down hardware requirements of the targeted MCUs and even to enable the extension of already deployed devices. We demonstrated practicability and efficiency of implementing our scheme on two representative MCUs and proposed extensions for usage in real-world scenarios. For future work, we will investigate support of additional device types, to widen to scope of applicability. A second effort will be taken to refine existing and develop further protocol extensions, such as symmetric sealing of assets (i.e., sensor values, etc.), establishment of trusted channels or means to log provenance of such assets, especially if they are computed on flash-based media that employ or scheme.

Acknowledgments. This work has been partly funded by the DFG as part of project P3 within the CRC 1119 CROSSING and the LOEWE initiative (Hessen, Germany) within the NICER project. The authors would also like to thank the anonymous reviewers for their valuable comments.

References

1. Ambrosin, M., Conti, M., Ibrahim, A., Neven, G., Sadeghi, A.-R., Schunter, M.: SANA: secure and scalable aggregate network attestation. In: CCS. ACM (2016)
2. Anati, I., Gueron, S., Johnson, S., Scarlata, V.: Innovative technology for CPU based attestation and sealing. In: HASP (2013)
3. Armknecht, F., Maes, R., Sadeghi, A.-R., Sunar, B., Tuyls, P.: Memory leakage-resilient encryption based on physically unclonable functions. In: Towards Hardware-Intrinsic Security (2010)
4. Armknecht, F., Sadeghi, A.-R., Schulz, S., Wachsmann, C.: A security framework for the analysis and design of software attestation. In: CCS. ACM (2013)
5. Ahuja, A.: SPMA044A - Using Execute, Write, and Erase-Only Flash Protection on Stellaris Microcontrollers Using Code Composer Studio
6. Asokan, N., Brasser, F., Ibrahim, A., Sadeghi, A.-R., Schunter, M., Tsudik, G., Wachsmann, C.: SEDA: scalable embedded device attestation. In: CCS (2015)
7. Bösch, C., Guajardo, J., Sadeghi, A.-R., Shokrollahi, J., Tuyls, P.: Efficient helper data key extractor on FPGAs. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 181–197. Springer, Heidelberg (2008). doi:10.1007/978-3-540-85053-3_12
8. Brasser, F., El Mahjoub, B., Sadeghi, A.-R., Wachsmann, C., Koeberl, P.: TyTAN: tiny trust anchor for tiny devices. In: DAC (2015)
9. Carpent, X., ElDefrawy, K., Rattanavipanon, N., Tsudik, G.: Lightweight swarm attestation: a tale of two LISA-s. In: AsiaCCS. ACM (2017)
10. Castelluccia, C., Francillon, A., Perito, D., Soriente, C.: On the difficulty of software-based attestation of embedded devices. In: CCS. ACM (2009)
11. Chen, W., Bhadra, J., Wang, L.-C.: SoC security and debug. In: Bhunia, S., Ray, S., Sur-Kolay, S. (eds.) Fundamentals of IP and SoC Security, pp. 29–48. Springer, Cham (2017). doi:10.1007/978-3-319-50057-7_3

12. Costin, A., Zaddach, J., Francillon, A., Balzarotti, D.: A large-scale analysis of the security of embedded firmwares. In: USENIX Security (2014)
13. Eldefrawy, K., Tsudik, G., Francillon, A., Perito, D.: SMART: secure and minimal architecture for (establishing dynamic) root of trust. In: NDSS (2012)
14. Francillon, A., Nguyen, Q., Rasmussen, K.B., Tsudik, G.: A minimalist approach to remote attestation. In: DATE (2014)
15. Hern, A.: Chinese webcam maker recalls devices after cyberattack link, October 2016. https://www.theguardian.com/technology/2016/oct/24/chinese-webcam-maker-recalls-devices-cyberattack-ddos-internet-of-things-xiongmai. Accessed 19 Apr 2017
16. Karame, G.O., Li, W.: Secure erasure and code update in legacy sensors. In: Conti, M., Schunter, M., Askoxylakis, I. (eds.) Trust 2015. LNCS, vol. 9229, pp. 283–299. Springer, Cham (2015). doi:10.1007/978-3-319-22846-4_17
17. Koeberl, P., Schulz, S., Sadeghi, A.-R., Varadharajan, V.: TrustLite: a security architecture for tiny embedded devices. In: EuroSys (2014)
18. Kohnhäuser, F., Katzenbeisser, S.: Secure code updates for mesh networked commodity low-end embedded devices. In: Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds.) ESORICS 2016. LNCS, vol. 9879, pp. 320–338. Springer, Cham (2016). doi:10.1007/978-3-319-45741-3_17
19. Kohnhäuser, F., Schaller, A., Katzenbeisser, S.: PUF-based software protection for low-end embedded devices. In: Conti, M., Schunter, M., Askoxylakis, I. (eds.) Trust 2015. LNCS, vol. 9229, pp. 3–21. Springer, Cham (2015). doi:10.1007/978-3-319-22846-4_1
20. Kovah, X., Kallenberg, C., Weathers, C., Herzog, A., Albin, M., Butterworth, J.: New results for timing-based attestation. In: Security & Privacy (2012)
21. Krebs, B.: Who Makes the IoT Things Under Attack? October 2016. https://krebsonsecurity.com/2016/10/who-makes-the-iot-things-under-attack/. Accessed 19 Apr 2017
22. Li, Y., Cheng, Y., Gligor, V., Perrig, A.: Establishing software-only root of trust on embedded systems: facts and fiction. In: Christianson, B., Švenda, P., Matyáš, V., Malcolm, J., Stajano, F., Anderson, J. (eds.) Security Protocols 2015. LNCS, vol. 9379, pp. 50–68. Springer, Cham (2015). doi:10.1007/978-3-319-26096-9_7
23. Li, Y., McCune, J.M., Perrig, A.: SBAP: software-based attestation for peripherals. In: Acquisti, A., Smith, S.W., Sadeghi, A.-R. (eds.) Trust 2010. LNCS, vol. 6101, pp. 16–29. Springer, Heidelberg (2010). doi:10.1007/978-3-642-13869-0_2
24. Li, Y., McCune, J.M., Perrig, A.: VIPER: verifying the integrity of PERipherals' firmware. In: CCS. ACM (2011)
25. Linux Foundation: Intel Quark Microcontroller Software Interface. Accessed 19 Apr 2017
26. Linux Foundation: Zephyr Project. https://www.zephyrproject.org/. Accessed 19 Apr 2017
27. Maes, R., Tuyls, P., Verbauwhede, I.: Low-overhead implementation of a soft decision helper data algorithm for SRAM PUFs. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 332–347. Springer, Heidelberg (2009). doi:10.1007/978-3-642-04138-9_24
28. Trusted Computing Group: TPM Main Specification. http://www.trustedcomputinggroup.org/resources/tpm_main_specification. Accessed 19 Apr 2017
29. Nohl, K., Krißler, S., Lell, J.: BadUSB - On accessories that turn evil (2014). https://opensource.srlabs.de/projects/badusb. Accessed 19 Apr 2017

30. 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)
31. Parno, B., McCune, J.M., Perrig, A.: Bootstrapping Trust in Modern Computers. Springer, New York (2011)
32. Perito, D., Tsudik, G.: Secure code update for embedded devices via proofs of secure erasure. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 643–662. Springer, Heidelberg (2010). doi:10.1007/978-3-642-15497-3_39
33. Real Time Engineers Ltd.: FreeRTOS Website. Accessed 9 Dec 2015
34. Ryan, M.: Bluetooth: with low energy comes low security. In: WOOT (2013)
35. Saponas, T.S., Lester, J., Hartung, C., Agarwal, S., Kohno, T.: Devices that tell on you: privacy trends in consumer ubiquitous computing. In: USENIX Security (2007)
36. Schaller, A., Arul, T., van der Leest, V., Katzenbeisser, S.: Lightweight anti-counterfeiting solution for low-end commodity hardware using inherent PUFs. In: Holz, T., Ioannidis, S. (eds.) Trust 2014. LNCS, vol. 8564, pp. 83–100. Springer, Cham (2014). doi:10.1007/978-3-319-08593-7_6
37. Schneier, B.: The internet of things is wildly insecure and often unpatchable. Wired, January 2014
38. Schrijen, G.-J., van der Leest, V.: Comparative analysis of SRAM memories used as PUF primitives. In: DATE (2012)
39. Seshadri, A., Luk, M., Perrig, A., van Doorn, L., Khosla, P.: SCUBA: secure code update by attestation in sensor networks. In: WiSe (2006)
40. Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., Khosla, P.: Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In: SOSP (2005)
41. Seshadri, A., Perrig, A., van Doorn, L., Khosla, P.: SWATT: software-based attestation for embedded devices. In: Security & Privacy. IEEE (2004)
42. Shaked, Y., Wool, A.: Cracking the Bluetooth PIN. In: MobiSys (2005)
43. STMicroelectronics: Proprietary code read-out protection on microcontrollers of the STM32L4 series. Accessed 23 June 2017
44. Texas Instruments: Stellaris LM4F120 LaunchPad Evaluation Kit. http://www.ti.com/tool/ek-lm4f120xl. Accessed 19 Apr 2017
45. Texas Instruments: Crypto-Bootloader (CryptoBSL) for MSP430FR59xx and MSP430FR69xx MCUs. Accessed 23 June 2017