

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

Steffen Schulz<sup>1</sup>, André Schaller<sup>2</sup>, Florian Kohnhäuser<sup>2</sup>, and  
Stefan Katzenbeisser<sup>2</sup>

<sup>1</sup> Intel Labs

`steffen.schulz@intel.com`

<sup>2</sup> Security Engineering Group, TU Darmstadt  
`lastname@seceng.informatik.tu-darmstadt.de`  
CYSEC, Mornewegstrasse 32, 64293 Darmstadt

**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 [13]

and vendors have varying opinions about privacy and access control in this new space [23, 37], an arguably critical requirement for survivable IoT infrastructures is the capability to apply security patches and recover from compromises [16, 39].

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 [9, 31]. Bluetooth devices may offer an even larger attack surface, since typically employed security mechanisms were shown to be insufficient [36, 45]. 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 [22, 25, 44]. 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, 11, 24]. 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 [14, 15, 19, 32]. 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, which is verifying software integrity after firmware updates. These works often target device classes that cannot be secured using hardware-based attestation approaches, such as legacy and low-end devices [18, 20, 34, 41]. Nevertheless, they only address a subset of attestation usage, 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 well-suited to increase the security for resource-constrained embedded devices (cf. Section 2). In more detail:

*Boot Attestation concept:* Instead of recording measurements in a secure environment, as in traditional TPM-like attestation, software integrity measurements are immediately authenticated as the platform boots. We will argue in Section 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<sup>rd</sup> party verification:* In Section 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 Section 5, Boot Attestation side-steps hardware requirements that were deemed essential for remote attestation until now [14].

*Analysis and implementation:* In Section 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<sup>3</sup>, 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 Figure 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). The application is typically initialized by an immutable component, such as a boot ROM or firmware boot loader, which reduces the risk of permanently disabling a device (“bricking”). When programmed via low-level interfaces such as JTAG, many devices allow to customize this early stage(s) of boot. We will revisit this property when implementing our Root of Trust (RoT) in Section 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).

---

<sup>3</sup> We use the terms software and firmware interchangeably in this work.



$r \leftarrow \text{attest}_{\text{AK}}(c, M')$  at  $\mathcal{P}$ , where  $c$  is a random challenge and  $\text{AK}$  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  $(c, \text{AK})$  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 Section 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 software stage  $x$  immediately into an authenticated state  $M_x$ , before handing control to the next software stage. This way,  $m_x$  is protected from manipulations by any subsequently loaded software. 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  $\mathcal{V}$ :

$$M_x \leftarrow \text{PRF}_{\text{AK}}(M_{x-1}, m_x)$$

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

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

Note that we can instantiate PRF and KDF using a single HMAC. The measurement state  $M_x$  has become implicit in  $\text{AK}_x$  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 software state as it is possible with TPM PCRs. The next section expands this idea into a full remote attestation protocol.

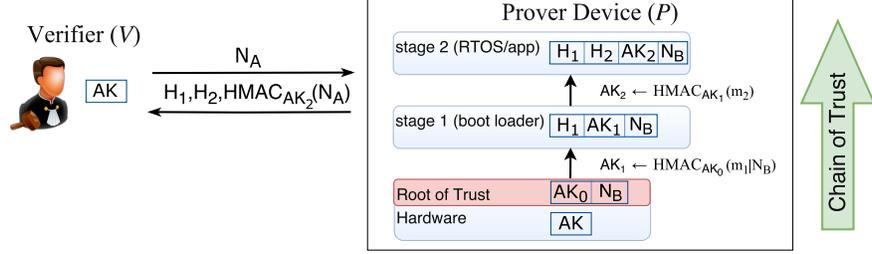


Fig. 2: Schematic overview of one possible instantiation of our Boot Attestation scheme as part of a remote attestation protocol.

### 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 software state by demonstrating possession of the implicitly authenticated measurement state  $AK_2$ .

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  $AK_0 \leftarrow AK$  and a boot nonce  $N_B$ . It derives  $AK_1$  as  $HMAC_{AK_0}(N_B, m_1)$ , with  $m_1 := (start_1, size_1, H_1)$  defined as the binary measurement of the software stage 1. Before launching stage 1, the RoT must purge intermediate secrets from memory and lock  $AK$  against further access by software. Execution then continues at stage 1 using the intermediate attestation key  $AK_1$  and measurement log  $(H_1, N_B)^4$ .

The scheme continues through other boot stages  $x \in \{1, \dots, k\}$  until the main application/runtime has launched in stage  $k$ . In each stage, a measurement  $m_{x+1}$  of the next software stage is taken and extended into the measurement state as  $AK_{x+1} \leftarrow HMAC_{AK_x}(m_{x+1})$ . 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 software 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 software state. For  $\mathcal{P}$ , this simply involves computing a proof of knowledge  $r \leftarrow HMAC_{AK_k}(N_A)$  and sending it to  $\mathcal{V}$  together with the measurement log. Using this response, the verifier  $\mathcal{V}$  can reconstruct the state  $M' = (m'_1, \dots, m'_k)$  claimed by  $\mathcal{P}$  and the associated  $AK_k$ .  $\mathcal{V}$  can then validate and accept  $\mathcal{P}$  if  $M' = M$  and  $r = HMAC_{AK_k}(N_A)$ .

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 entities  $(AK, M)$ .

<sup>4</sup> 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.

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 re-boot  $\mathcal{P}$  to refresh all  $AK_x$ . However, while the implications of managing  $N_B$  are discussed in Section 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 attestation game, defined in Section 2.3, and the adversary, specified in Section 2.2. 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 [30]. 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 considered adversary  $\mathcal{A}$  is able eavesdrop, synthesize, manipulate, and drop any network data. Boot Attestation, however, relies on a challenge-response protocol, involving the shared key  $AK$ , which is a standard defense against network 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  $r = HMAC_{AK_k}(N_A)$ . Replay is mitigated using a fresh nonce  $N_A$  in each attestation. Dropping messages only results in a generic DoS, such that  $\mathcal{V}$  will not accept  $\mathcal{P}$ . Furthermore, 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, 10].

**Physical and Side-Channel Attacks.**  $\mathcal{A}$  may also attempt simple hardware attacks on  $\mathcal{P}$ . In particular,  $\mathcal{A}$  may use SoC-external interfaces, such as power supply and debug interfaces, to gain information on intermediate attestation keys  $AK_x$ , or manipulate SoC-external memory and I/O. The extent of hardware attack resilience mostly depends on the SoC implementation and is out of scope. Boot Attestation assumes basic hardware mechanisms to protect debug ports and prevent leakage of intermediate computation states (cf. Section 5).

In addition,  $\mathcal{A}$  could still try to perform software side-channel attacks, such as cache, data remanence, or time attacks. Yet, since each stage  $i$  cleans any data besides  $N, H_{i+1}, AK_{i+1}$ , 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 non-paged, tightly coupled SRAM employed in the RoT is straightforward.

**Load-time Compromise.**  $\mathcal{A}$  may compromise the software stage  $a$  of  $\mathcal{P}$  before it is loaded and, hence, measured. In this case,  $\mathcal{A}$  can access all intermediate

measurements  $(m_1, \dots, m_k)$ , the nonces  $(N_B, N_A)$ , and any subsequent attestation keys  $(AK_a, \dots, AK_k)$ . Note that compromising the RoT (i.e., the initial stage) is outside the capabilities of  $\mathcal{A}$ , which is a reasonable assumption due to RoT’s hardware protection (see Section 5) and its miniscule code complexity (see Table 2).

Compromising the intermediate measurement state and keys allows  $\mathcal{A}$  to build 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 performed by  $\mathcal{A}$  and cannot be expanded to spoof a valid software state  $M_k$  anymore.

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 software 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 foresee this renewed key chain, as this would require access to at least  $AK_{a-1}$ .

**Runtime Compromise.**  $\mathcal{A}$  may also compromise the software in stage  $a$  at runtime after it is measured, e.g., by exploiting a software vulnerability that leads to arbitrary code execution. In this case,  $\mathcal{A}$  has access to the correct (unmodified) attestation key  $AK_a$ , is able to bypass the chain of trust, and thus is able to win the attestation game. We would like to point out that all load-time attestation schemes, e.g. the TPM [30], are vulnerable to such runtime attacks. Even if  $\mathcal{P}$  performs a reboot, measures the compromised software in stage  $a$ , and thus reconstructs an incorrect (modified) attestation key  $AK'_a$ , our scheme is still broken. This is because  $\mathcal{A}$  may record  $AK_a$  somewhere and reuse it after reboot to win any attestation game.

Nevertheless,  $\mathcal{V}$  is able to patch the attestation keys  $AK_1, \dots, AK_k$  secure again by setting a new boot nonce  $N_B$ , as introduced in Section 3.2, and rebooting the device. This way, completely new attestation keys are valid, such that the compromised  $AK_a$  becomes invalid. Note that  $N_B$  may also be implemented as a timestamp coming from a write-protected real-time clock (RTC) that can only be set by  $\mathcal{V}$ . Even more security provides the attestation key provisioning approach, described in Section 4.1, which additionally patches  $AK_0$  secure again. Another option is to store and develop  $AK_x$  in a hardware crypto accelerator, such that it cannot be accessed directly by software. However, although this is a standard component only a small minority of COTS MCUs support hardware crypto acceleration today.

## 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 (see Section 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 [14, 32], 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 verifiers  $\mathcal{V}'$ , 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 manufacturer and only distribute and use valid public keys based on expected firmware measurements. The detailed protocol works as follows:

Initially,  $\mathcal{P}$  and  $\mathcal{CA}$  share a common secret  $AK$  and that  $\mathcal{P}$  was initialized according to Section 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  $(AK_{priv}, AK_{pub}) \leftarrow KeyGen(AK_k)$ .

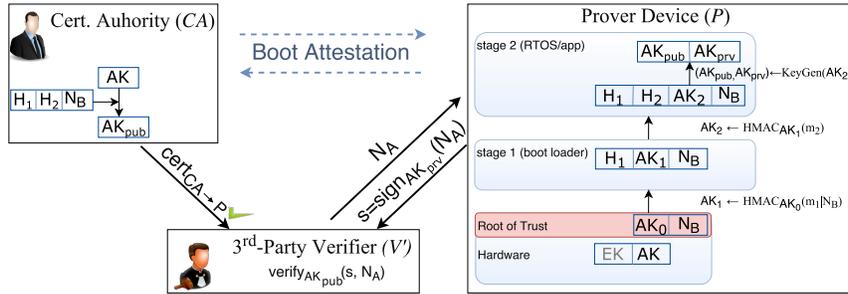


Fig. 3: Third-party verification using a trusted  $\mathcal{CA}$ . The optional boot attestation phase is depicted with dashed arrows.

This can be done deterministically for example using ECC key generation. Subsequently,  $\mathcal{CA}$  receives  $(H_1, \dots, H_k, N, r = \text{HMAC}_{\text{AK}_k}(c))$  from  $\mathcal{P}$ . Using the intermediate hashes  $(H_1, \dots, H_k)$  and  $\text{AK}$ ,  $\mathcal{CA}$  can reproduce  $\text{AK}_k$  and  $\mathcal{P}$ 's public key  $\text{AK}_{\text{pub}}$  and publish a certificate  $\text{cert}_{\mathcal{CA} \rightarrow \mathcal{P}} \leftarrow \text{Sign}_{\mathcal{CA}_{\text{priv}}}(\text{AK}_{\text{pub}})$ .

The third party  $\mathcal{V}'$  initiates attestation by querying  $\mathcal{CA}$  for  $\mathcal{P}$ 's signed public key  $\text{cert}_{\mathcal{CA} \rightarrow \mathcal{P}}$ . 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  $N_A$   $s \leftarrow \text{Sign}_{\text{AK}_{\text{priv}}} (N_A)$  and sends  $s$  to  $\mathcal{V}'$ . The third party is now able to infer statements about  $\mathcal{P}$ 's identity and software state. At the same time  $\text{AK}_k$  is kept secret from  $\mathcal{V}'$ . An overview of the scheme is shown in Figure 3.

## 5 Hardware Requirements

We are now ready to describe the detailed hardware requirements of the boot attestation scheme. We formulate these as results here and not as system assumptions in Section 2.1, since the exploration of alternative remote attestation schemes with minimal hardware requirements has been a major research challenge in recent years [14, 15, 19, 32]. 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 [11, 24].

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  $\text{AK}$ . 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 exclusively available to the RoT. 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. Section 3.3).

**[II\*] AK Provisioning:** Provisioning of a new attestation key  $AK_{new}$  involves replacement of its previous instance, conducted by the RoT (cf. Section 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. Section 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 [12]. 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-programming 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 [14, 15]. Based on these hardware requirements, we will now discuss possible implementation options and present evaluation results.

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

Device Type	CPU	SRAM (kB) / Flash (kB) / EEPROM (B)	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	✓
MSP430F5308	MSP430	6 / 16 / -	Flash	Flash	Flash	✓
LPC1200	Cortex-M0	4-8 / 32-128 / -	Flash	✗	✗	✓
STM32F100Rx	Cortex-M3	8 / 64-128 / -	Flash	✗	PUF	✗
<i>Stellaris LM4F120</i>	Cortex-M4F	32 / 256 / 2048	Flash	Flash	Flash/EEPROM	✓
<i>Quark MCU D2000</i>	Quark D2000	8 / 44 / -	Flash	Flash (main)	Flash (OTP)	✓
Arduino/Genuino101	Quark SE C1000	80 / 392 / -	Flash	Flash	Flash	✓

## 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, 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\*])<sup>5</sup>. 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.

<sup>5</sup> Note that if no AK provisioning is desired, EK protection support is sufficient to store the root attestation key (requirement [II]).

## 6.1 Prototype I: TI Stellaris LaunchPad

The TI Stellaris Launchpad [47] implements an ARM Cortex-M4F operating at 80 MHz<sup>6</sup>, 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 appliances. We use FreeRTOS [35] as a software 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 & EK.** Although the Stellaris provides memory protection for flash [5, 46], 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 [29, 40]. Indeed, the SRAM-based PUF instance of the Stellaris has already been characterized in [21]. 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 [38]. 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.

---

<sup>6</sup> In our experiments we set the operating frequency to 50 MHz to allow for better comparison with the Intel MCU.

## 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) [27] and Zephyr RTOS [28] as the standard software stack.

**Integrity Protected RoT & Debug Protection.** The D2000 boots directly from an 8 kB OTP flash partition. An 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 & 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 the most memory-efficient setting, the scheme requires a total of  $\approx 3.1$  kB on the Stellaris. This may seem large compared to the 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.

	Size (Bytes)				Runtime (ms)			
	ARM		x86		ARM		x86	
Component	-0s	-01	-0s	-01	-0s	-01	-0s	-01
Base ROM	702	712	1955	2115	0.79	0.63	6.11	5.93
Root of Trust (RoT)								
<i>Base Logic</i>	336	340	168	193	< 0.01	< 0.01	< 0.01	< 0.01
<i>HMAC-SHA2 (256 bit)</i>	1828	1836	1819	2061	3.04	3.04	1.54	1.44
<i>HMAC-SHA2 (32 kB)</i>	1828	1836	1819	2061	312.26	312.26	148.23	145.37
AK Protection								
<i>Flash</i>	—	—	295	337	—	—	0.02	0.02
<i>EEPROM</i>	516	580	—	—	0.01	< 0.01	—	—
EK Protection								
<i>Flash</i>	—	—	378	448	—	—	< 0.01	0.002
<i>EEPROM</i>	516	580	—	—	0.01	< 0.01	—	—
<i>SRAM PUF</i>	1662	1980	—	—	46.44	46.42	—	—

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 (-0s) and runtime (-01).

**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*.

Hardware-based attestation schemes rely on secure hardware, such as Intel SGX or a Trusted Platform Module (TPM) [2, 30], 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 [14, 17, 19, 32, 15], 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 [22, 26, 42, 43]. However, they rely on assumptions that have proven to be hard to achieve in practice [4, 11, 24]. 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 (so-called device swarms) that are interconnected in large mesh networks [1, 6, 10]. 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, 14, 19], they could leverage our Boot Attestation scheme to be applicable to a broader range of embedded devices.

Secure code updates specifically addresses a subset of remote attestation, namely, verifying the integrity of software after it is updated. Initial approaches build upon software-based attestation techniques to perform secure code updates [41]. Accordingly, they inherit both the good and bad qualities of software-based techniques, i.e., being theoretically applicable to any device-class but relying on unrealistic timing assumptions (see last paragraph). Later on, the notion of Proofs of Secure Erasure (PoSE) was introduced to secure code updates [18, 34]. PoSE-based approaches neither rely on tight timing assumptions nor on secure hardware. Instead, they apply a challenge-response protocol that requires the prover to fill its entire memory with data, such that any malicious code is overwritten. PoSE-based approaches can be applied to many devices, as they only require a small amount of read-only memory (ROM) on the prover. However, they assume an adversary who is present in the network to be able to only communicate with the verifier and not the prover device, which is a strong limitation. Recently, another secure code update scheme was proposed, which focuses on scalable updates in large mesh networks [20]. In contrast to our work, it imposes the use of asymmetric cryptography, involving heavy computational overhead and a large memory footprint.

## 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 feasibility and efficiency of our scheme on commercial hardware 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.

## 9 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.

## References

1. M. Ambrosin, M. Conti, A. Ibrahim, G. Neven, A.-R. Sadeghi, and M. Schunter. SANA: Secure and Scalable Aggregate Network Attestation. In *Conference on Computer & Communications Security*, pages 731–742, 2016.
2. I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, 2013.
3. F. Armknecht, R. Maes, A.-R. Sadeghi, B. Sunar, and P. Tuyls. Memory leakage-resilient encryption based on physically unclonable functions. In *Towards Hardware-Intrinsic Security*, pages 135–164. 2010.
4. F. Armknecht, A.-R. Sadeghi, S. Schulz, and C. Wachsmann. A security framework for the analysis and design of software attestation. In *Conference on Computer & Communications Security*, pages 1–12, 2013.
5. Ashish Ahuja. SPMA044A – Using Execute, Write, and Erase-Only Flash Protection on Stellaris Microcontrollers Using Code Composer Studio.
6. N. Asokan, F. Brasser, A. Ibrahim, A.-R. Sadeghi, M. Schunter, G. Tsudik, and C. Wachsmann. Seda: Scalable embedded device attestation. In *Conference on Computer & Communications Security*, pages 964–975, 2015.
7. C. Bösch, J. Guajardo, A.-R. Sadeghi, J. Shokrollahi, and P. Tuyls. Efficient helper data key extractor on FPGAs. In *Cryptographic Hardware and Embedded Systems*, pages 181–197. 2008.
8. F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl. Tytan: tiny trust anchor for tiny devices. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2015.
9. bunnief and xobs at 30C3. SD Card Hacking. <http://www.bunniefoo.com/bunnief/sdcard-30c3-pub.pdf>. Last accessed 19th April 2017.
10. X. Carpent, K. ElDefrawy, N. Rattanavipanon, and G. Tsudik. Lightweight Swarm Attestation: A Tale of Two LISA-s. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 86–100, 2017.
11. C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *Conference on Computer & Communications Security*, pages 400–409, 2009.
12. W. Chen, J. Bhadra, and L.-C. Wang. SoC Security and Debug. In *Fundamentals of IP and SoC Security*, pages 29–48. Springer, 2017.
13. A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A Large-Scale Analysis of the Security of Embedded Firmwares. In *USENIX Security*, pages 95–110, 2014.

14. K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *NDSS*, volume 12, pages 1–15, 2012.
15. A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik. A Minimalist Approach to Remote Attestation. In *Design, Automation & Test in Europe*, 2014.
16. A. Hern. Chinese webcam maker recalls devices after cyberattack link. <https://www.theguardian.com/technology/2016/oct/24/chinese-webcam-maker-recalls-devices-cyberattack-ddos-internet-of-things-xiongmai>, Oct. 2016. Last accessed 19th April 2017.
17. G. Hunt, G. Letey, and E. Nightingale. The seven properties of highly secure devices. Technical report, March 2017.
18. G. O. Karame and W. Li. Secure Erasure and Code Update in Legacy Sensors. In *Trust and Trustworthy Computing*, pages 283–299. 2015.
19. P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. TrustLite: A Security Architecture for Tiny Embedded Devices. In *European Conference on Computer Systems*, pages 10:1–10:14, 2014.
20. F. Kohnhäuser and S. Katzenbeisser. Secure Code Updates for Mesh Networked Commodity Low-End Embedded Devices. In *European Symposium on Research in Computer Security*, pages 320–338, 2016.
21. F. Kohnhäuser, A. Schaller, and S. Katzenbeisser. PUF-Based Software Protection for Low-End Embedded Devices. In *Trust and Trustworthy Computing*. 2015.
22. X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth. New results for timing-based attestation. In *Security & Privacy*, 2012.
23. B. Krebs. Who Makes the IoT Things Under Attack? <https://krebsonsecurity.com/2016/10/who-makes-the-iot-things-under-attack/>, Oct. 2016. Last accessed 19th April 2017.
24. Y. Li, Y. Cheng, V. Gligor, and A. Perrig. Establishing software-only root of trust on embedded systems: Facts and fiction. In *International Workshop on Security Protocols*, pages 50–68. Springer, 2015.
25. Y. Li, J. M. McCune, and A. Perrig. SBAP: Software-Based Attestation for Peripherals. In *Trust and Trustworthy Computing*, pages 16–29, 2010.
26. Y. Li, J. M. McCune, and A. Perrig. VIPER: verifying the integrity of PERipherals’ firmware. In *Conference on Computer & Communications Security*, 2011.
27. Linux Foundation. Intel Quark Microcontroller Software Interface. Last accessed 19th April 2017.
28. Linux Foundation. Zephyr Project. <https://www.zephyrproject.org/>. Last accessed 19th April 2017.
29. Maes, Roel and Tuyls, Pim and Verbauwhede, Ingrid. Low-overhead implementation of a soft decision helper data algorithm for SRAM PUFs. In *Cryptographic Hardware and Embedded Systems*, pages 332–347. 2009.
30. Trusted Computing Group. TPM Main Specification. [http://www.trustedcomputinggroup.org/resources/tpm\\_main\\_specification](http://www.trustedcomputinggroup.org/resources/tpm_main_specification). Last accessed 19th April 2017.
31. K. Nohl, S. Kriler, and J. Lell. BadUSB – On accessories that turn evil. <https://opensource.srlabs.de/projects/badusb>, 2014. Last accessed 19th April 2017.
32. J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base. In *USENIX Security*, pages 479–494, 2013.
33. B. Parno, J. M. McCune, and A. Perrig. *Bootstrapping Trust in Modern Computers*. Springer, August 2011.

34. D. Perito and G. Tsudik. Secure Code Update for Embedded Devices via Proofs of Secure Erasure. In *ESORICS*, volume 6345, pages 643–662. Springer, 2010.
35. Real Time Engineers Ltd. FreeRTOS Website. Last accessed 9th December 2015.
36. M. Ryan. Bluetooth: With Low Energy Comes Low Security. In *WOOT*, 2013.
37. T. S. Saponas, J. Lester, C. Hartung, S. Agarwal, and T. Kohno. Devices that tell on you: Privacy trends in consumer ubiquitous computing. In *USENIX Security*, pages 5:1–5:16. USENIX, 2007.
38. A. Schaller, T. Arul, V. van der Leest, and S. Katzenbeisser. Lightweight Anti-counterfeiting Solution for Low-End Commodity Hardware Using Inherent PUFs. In *Trust and Trustworthy Computing*, pages 83–100. Springer, 2014.
39. B. Schneier. The Internet of Things Is Wildly Insecure And Often Unpatchable. *Wired*, Jan, 2014.
40. G.-J. Schrijen and V. van der Leest. Comparative analysis of SRAM memories used as PUF primitives. In *Conference on Design, Automation and Test in Europe*, pages 1319–1324. EDA Consortium, 2012.
41. A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla. Scuba: Secure code update by attestation in sensor networks. In *ACM workshop on Wireless security*, pages 85–94. ACM, 2006.
42. A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. *Operating Systems Review*, 39(5):1–16, 2005.
43. A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla. Swatt: Software-based attestation for embedded devices. In *Security & Privacy*, pages 272–282. IEEE, 2004.
44. A. Seshadri, A. Perrig, L. van Doorn, and P. K. Khosla. SWATT: software-based attestation for embedded devices. In *Security and Privacy*, page 272. IEEE, 2004.
45. Y. Shaked and A. Wool. Cracking the Bluetooth PIN. In *Mobile systems, applications, and services*, pages 39–50. ACM, 2005.
46. Texas Instruments. Stellaris LM3S9B96 Microcontroller – Data Sheet.
47. Texas Instruments. Stellaris LM4F120 LaunchPad Evaluation Kit. <http://www.ti.com/tool/ek-lm4f120x1>. Last accessed 19th April 2017.