Secure Code Updates for Smart Embedded Devices based on PUFs

Wei Feng, Yu Qin, Shijun Zhao, and Dengguo Feng

Trusted Computing and Information Assurance Laboratory,
Institute of Software Chinese Academy of Sciences
vonwaist@gmail.com

Abstract. Code update is a very useful tool commonly used in low-end embedded devices to improve the existing functionalities or patch discovered bugs or vulnerabilities. If the update protocol itself is not secure, it will only bring new threats to embedded systems. Thus, a secure code update mechanism is required. However, existing solutions either rely on strong security assumptions, or result in considerable storage and computation consumption, which are not practical for resource-constrained embedded devices (e.g., in the context of Internet of Things). In this work, we first propose to use intrinsic device characteristics (i.e., Physically Unclonable Functions or PUF) to design a practical and lightweight secure code update scheme. Our scheme can not only ensure the freshness, integrity, confidentiality and authenticity of code update, but also verify that the update is installed correctly on a specific device without any malicious software. Cloned or counterfeit devices can be excluded as the code update is bound to the unpredictable physical properties of underlying hardware. Legitimate devices in an untrustworthy software state can be restored by filling suspect memory with PUF-derived random numbers. After update installation, the initiator of the code update is able to obtain the verifiable software state from device, and the device can maintain a sustainable post-update secure check by enforcing a secure call sequence. To demonstrate the practicality and feasibility, we also implement the proposed scheme on a low-end MCU platform (TI MSP430) by using onboard SRAM and Flash resources.

Keywords: Firmware Update, Secure Code Update, Physically Unclonable Function (PUF), Remote Attestation, Embedded Security

1 Introduction

With the rise of new trends like the Internet of Things (IoT), Industry 4.0, or Industrial Internet, smart embedded devices are being increasingly used in various scenarios, such as industrial control, smart home, wireless sensor networks, etc. Firmware or code update is an important mechanism for these scenarios as it offers many benefits [31, 35]: fix bugs or vulnerabilities that have been disclosed in the deployed devices; add new features or functionalities to system; enable or disable product functionality in the field; reduce the number of product returns

to be handled. For example, as recently reported, Dyn DNS DDoS attack[1] is caused by a large number of IoT botnet nodes infected with the Mirai malware. Code update mechanisms may be used to repair these embedded nodes without having to recall or destroy these devices. However, if the update process itself is vulnerable, it can be exploited by attackers to compromise the security of embedded systems. As low-end embedded devices are resource-constrained and often lack the security capabilities of general purpose computing platforms, it's difficult and challenging to design a secure code update mechanism for them.

A secure code update scheme for embedded systems should not only consider a protocol for secure downloading, but also ensure that the newly downloaded code is installed properly and its memory can be verified with the confidence that no attacker or malicious code is involved. Ideally, a secure code update should provide the following security attributes [31, 35]: (1) **Freshness**, the downloaded code is newest, not a simple replay or downgrading; (2) **Integrity**, the update code installed on device is expected and unmodified; (3) **Authenticity**, the update code comes from an authorized source and is loaded onto an authorized device (cloning can also be detected), i.e., mutual authentication is needed; (4) **Confidentiality**, the code may be an important intellectual property, which should not be revealed to other parties; (5) **Feasibility**, the scheme is applicable to existing commodity low-end embedded devices based on existing resources; (6) **Verifiability**, after update installation, the software state of the updated device should be verified and the verification result should be eventually fed back to the source who issues the update; (7) **Restorability**, secure code update is able to restore the software state of a compromised device; and (8) **Secure Call**, only trustworthy code can be called and executed on the device after the update process is complete, which aims to alleviate TOCTOU attack [14].

Currently, there are few solutions that can satisfy all these attributes. By pointing out the inadequacies of existing techniques (hardware and software-based attestation), Perito and Tsudik [42] introduced a new notion called Proofs of Secure Erasure (PoSE) for secure code update, in which new code is downloaded onto an embedded device after secure erasure of all its prior state. PoSE meets the integrity, feasibility and restorability attributes. However, other security attributes are not supported. Furthermore, PoSE relies on strong security assumptions [42], e.g., the adversary maintains complete communication silence during attestation, and it also results in considerable energy and time overhead. The follow-up researches [17, 32] of PoSE all focus on reducing the communication and computation overhead, and rarely consider to improve the assumptions or strengthen security guarantees. Recently, Kohnhauser et al. [35] proposed a novel secure code update scheme for mesh networked embedded devices, which achieves much stronger security guarantees and satisfies most of the security attributes. Their method relies on three hardware security requirements: immutable code, secure storage and uninterruptible execution. Nevertheless, their method has the following flaws: (1) it uses the traditional secure storage technology (like EEPROM, BBRAM or eFuse) for device secret or private keys,

[1] `https://en.wikipedia.org/wiki/2016_Dyn_cyberattack`

which is expensive, inflexible and unsafe [39, 53]; (2) it uses public-key cryptography, which results in apparent storage consumption (66KB for signature) and increased running time; and (3) it is vulnerable to device cloning attack and TOCTOU attack.

Contributions. In this paper, we propose the first secure code update scheme for current commodity low-end embedded devices by using Physically Unclonable Functions (PUFs). Firstly, our scheme reserves the design of secure erasure from PoSE; however, the prover does not need to download random data as large as its own memory from the verifier. As an improvement, we fill the prover's memory with high entropy data derived from PUF. Additionally, we don't rely on the strong security assumption like communication silence. Secondly, as opposed to the latest method in ESORICS 2016 [35], we use PUF-based secure key generation to replace traditional secure key storage, and use symmetric cryptography and message authentication code (MAC) instead of public-key cryptography to achieve confidentiality and authenticity. Thirdly, we design a secure code update protocol based on reverse fuzzy extractor, which satisfies all the security attributes mentioned above. To illustrate this, we conclude eight possible security threats that may break these security attributes and show how our scheme can be used to address them. Finally, we implement and evaluate our protocol building blocks in a low-cost and general-purpose MSP430 MCU. The evaluation results demonstrate the feasibility and validity of PUF-based secure code update in low-end embedded devices.

Outline. In Section 2, we conclude the security threats and present some background knowledge about PoSE and PUF. In Section 3, we first give the system requirements and adversary model, and then introduce our new proposal for secure code update by using PUF. In Section 4, we implement and evaluate the building blocks of our novel scheme using a MSP430 device. In Section 5, we overview the related work, and we conclude the paper in Section 6.

2 Background and Preliminaries

2.1 Security Threats of Code Update

Code update involves a verifier V and a prover P. P is a generic embedded device with constrained resources, e.g. a medical instrument, a wearable device or an industrial control device. V is a more powerful computing device, e.g. a smartphone, a laptop or a cloud platform. Secure code update can be viewed as a means to ensure that a code update issued by a trusted V has been securely distributed and correctly installed on P. Specifically, for secure code update, we aim to provide measures to solve the following security threats [31, 21, 35]:

Threat-1 **Code Alteration**. The binary code (or firmware image) distributed by V is modified by attackers during the update process.

Threat-2 **Code Reverse Engineering**. Attackers intercept the binary image code, and use the reverse engineering technique to analyze the functionality and contents of the update image code.

Threat-3 **Loading Unauthorized Code**. The update binary code may be created by an unauthorized party, and P is cheated to install the unauthorized or malicious code.

Threat-4 **Loading Code onto an Unauthorized Device**. The code intended for one device is installed on another, or the code generated by the product manufacturer is loaded onto an unauthorized device.

Threat-5 **Code Downgrading**. An attacker in possession of an old code package may resend it to the device reverting it to a previous, possibly vulnerable, state in order to exploit it.

Threat-6 **Incomplete Update**. A compromised device may simply deny the execution of code update or execute it inappropriately without restoring software integrity. And at the same time, V is cheated with a response indicating a successful update.

Threat-7 **TOCTOU (Time Of Check to Time Of Use)**. After a complete update, the update code stored in the device may have been tampered with when it's called to run a specific embedded task.

Threat-8 **Device Cloning**. Attackers may simply copy the memory contents (including code, data, secrets or keys, and other intellectual property) and create a cloned device to replace the original one.

2.2 Proofs of Secure Erasure (PoSE)

While hardware-based attestation [8, 41] is not practical for low-cost embedded systems and software-based attestation [47] offers unclear security guarantees, Perito and Tsudik [42] proposed a new technique called *Proofs of Secure Erasure* (PoSE) for low-end embedded devices.

According to [42], PoSE can be used to implement a secure code update protocol, which we conclude in Figure 1. Suppose the size of P's memory (all writable storage on the embedded device) is n, the verifier V first encrypts the update code using a random key K'. Upon receiving the ciphertext blocks $\{R_1, ..., R_n\}$, P uses the last k-blocks of randomness as the key to compute a MAC (Message Authentication Code) and sends the MAC to V. V verifies the MAC to ensure that P's memory is reliably erased with the high entropy data (ciphertext) sent by V. If MAC verified correctly, V sends the encryption key K' to P in order for P to decrypt the ciphertext into the new code $\{C_1, ..., C_{n-k}\}$.

Overhead of PoSE. PoSE results in considerable communication and computation costs on the prover: the ciphertext transmitted from V to P is as large as P's memory, and P needs to compute a MAC over the entire memory. Therefore, Dziembowski et al. [17] constructed a new cryptographic primitive, called uncomputable hash functions, which can be used to improve the communication complexity of PoSE. Recently, Karame and Li [32] combined PoSE with All or Nothing Transforms to reduce the communication and computation overhead.

Security of PoSE. PoSE can resist Threat-1 through MAC computation, but cannot address all other threats while facing a public communication channel and an untrusted embedded system. By observing this, Kohnhauser and Katzenbeisser [35] provided a verifiable code update mechanism based on three

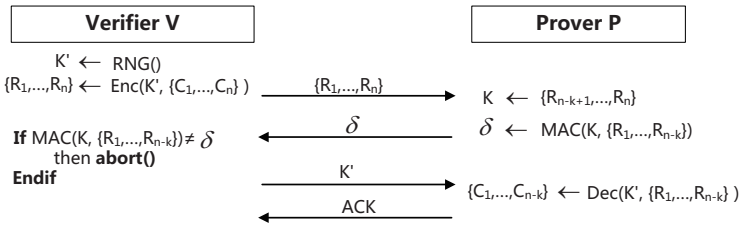

Fig. 1. Secure Code Update Protocol based on PoSE

hardware security requirements (immutable code, secure storage and uninterruptible execution), which is still vulnerable to device cloning and TOCTOU attacks. To solve all aforementioned security threats, we will propose a novel secure code update protocol without relying on secure storage by combing PoSE with SRAM PUF.

2.3 Physically Unclonable Function and Reverse Fuzzy Extractor

A physically unclonable function (PUF) is an entity that uses manufacturing variation to generate a device-specific output, which can be seen as the *fingerprint* of a device [11]. Specifically [44], when queried with a challenge C_i, a PUF generates a response $R_i = PUF(C_i)$ that depends on both, C_i and the unique IC intrinsic physical properties of the device containing PUF. The tuples (C_i, R_i) are thereby termed the challenge-response pairs (CRPs) of the PUF. PUFs offer security attributes such as uniqueness, reproducibility and unclonability; and can be used to implement some basic security applications: identification, authentication and key generation [11].

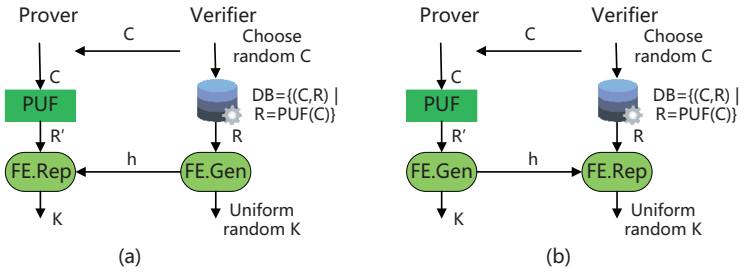

Fig. 2. Reproducible and uniformly distributed key from PUF. (a) Fuzzy Extractor. (b) Reverse Fuzzy Extractor.

Fuzzy Extractor and Reverse Fuzzy Extractor. PUFs are inherently noisy and their responses are not uniformly random, thus some mechanisms are needed to correct noise and extract randomness from the PUF responses.

Depending on the computing power of the prover device, there are two different mechanisms: fuzzy extractor (FE) and reverse fuzzy extractor (RFE). Fuzzy extractor [16] allows one to extract some randomness K from response R and then successfully reproduce K from any other response R' that is close to R. As shown in Figure 2(a), FE has two procedures: in the *generation procedure*, $FE.Gen$ on input a response R outputs an uniform random K and a helper data h; later in the *reproduction procedure*, $FE.Rep$ uses the helper data to recover $K = FE.Rep(R', h)$ from a distorted PUF response $R' = R + e$, where e is the error caused by noise. The security property of FE guarantees that the produced K is nearly uniform random even for those who observe the help data h, which means that h need not remain secret and can be stored and used publicly. An important thing to note is that the two algorithms $FE.Gen$ and $FE.Rep$ have asymmetric complexity [23, 10]: $FE.Rep$ (often containing error decoding algorithms) typically has higher complexity than $FE.Gen$. Apparently, $FE.Rep$ is not suitable for a constrained embedded device. To overcome this problem, van Herrewege et al. [23] proposed reverse fuzzy extractors (RFE), which place the $FE.Gen$ within the constrained prover device and move $FE.Rep$ to the more powerful verifier (Figure 2(b)).

As the prover in our system is a constrained embedded device, we adopt RFE to extract reconstructible random keys from PUF. Our RFE construction is based on a code-offset secure sketch and a strong random extractor described in [16]. The secure sketch uses the error correction algorithm to recover the raw PUF response. Formally, $FE.Gen$ and $FE.Rep$ are represented as follows:

$$
\begin{aligned}
&(K, h) \leftarrow FE.Gen(R') : \\
&\{r \leftarrow RNG(), CW \leftarrow Encode(r), h \leftarrow R' \oplus CW, K \leftarrow Ext(R')\} \\
&K \leftarrow FE.Rep(R, h) : \\
&\{CW' \leftarrow R \oplus h, r \leftarrow Decode(CW'), CW \leftarrow Encode(r), K \leftarrow Ext(CW \oplus h)\}
\end{aligned}
$$

Encode and *Decode* are two procedures included in the error correction. The random extractor (Ext) is used to obtain a full-entropy key K from PUF response. A random number generator RNG is used to choose a random codeword (CW), and CW only serves for error correction.

3 Secure Code Update based on PUF

3.1 System Requirements and Adversary Model

Our system consists of (at least) two players: a verifier V and a resource-constrained prover P. We denote the adversary with $\mathcal{A}$. The main goal is to allow V to update the application code of P, while providing effective measures to mitigate all kinds of security threats mentioned above.

Verifier V. We assume V to be trusted. Further, V initializes and deploys P in a secure environment, extracts adequate (at least two) challenge-response pairs (CRPs) from the PUF of P and stores them securely. V also keeps a copy of the update's binary code generated by the product manufacturer of P.

Prover P. We assume P to be equipped with a root of trust (RoT), which contains *a robust and unpredictable PUF*, *a reverse fuzzy extractor*, *a random number generator*, *a symmetric cryptographic algorithm*, *a secure one-way hash function* and *a message authentication code*. We also assume P has a static non-volatile write-protected memory region $\mathcal{R}$, which can be implemented based on Flash memory with dedicated lock bits as described in [35]. We assume the RoT code is stored in the protected region $\mathcal{R}$ isolated from the application code, and once the RoT code in $\mathcal{R}$ gets executed, it cannot be interrupted until the control flow intentionally leaves $\mathcal{R}$. The difference from [35] is that $\mathcal{R}$ here doesn't rely on a traditional secure storage, which is replaced by PUF-based key generation. We also assume the protection on $\mathcal{R}$ can be temporarily removed by RoT during the update and is restored immediately after the update, which can also be implemented on existing commercial embedded devices as described in [30]. It is worth noting that the update of RoT code itself (infrequently) should be offline in a secure environment.

Adversary $\mathcal{A}$. We assume that $\mathcal{A}$ has complete control over the communication channel between V and P. This means that $\mathcal{A}$ can eavesdrop, manipulate and reroute all messages sent between V and P. We assume $\mathcal{A}$ cannot clone or tamper the PUF feature of P. Following the typical assumptions on PUF-based key generation (like [10, 23]), we assume that $\mathcal{A}$ cannot access the challenge-response interface of PUF and cannot obtain temporary data (such as PUF-derived key information) stored in registers or on-device RAM during the update protocol. The temporary data can be erased by RoT immediately after the update protocol. In addition, we assume that $\mathcal{A}$ can be physically present and introduce additional (cloned) prover device. Finally, we assume $\mathcal{A}$ cannot bypass any of the hardware protections and cryptographic algorithms used in P. Data remanence attacks and physical attacks are not considered in our mechanism. We assume RoT code is immune from vulnerabilities, but the application code may be vulnerable. The device debug interfaces are disabled after deployment.

3.2 Update Protocol

Our new code update protocol is described in Figure 3, and the memory layout of P during the protocol execution is illustrated in Figure 4. V prepares a code update package *cupkg*, which includes (at least) the binary update code (*cupkg.code*), the current package version number (*cupkg.ver*), and the hash values over the expected memory contents for a successful update (*cupkg.hash*). P stores the RoT code and the expected integrity data (consisting of *cupkg.ver* and *cupkg.hash*) in the protected region $\mathcal{R}$, and the integrity data can also be updated securely by RoT during the protocol. Two PUF CRPs are used in the protocol, one for encryption and the other for mutual authentication.

Before each update protocol, we assume a temporary session key (tsk) is established between V and P by using a key exchange protocol (e.g., Diffie-Hellman or ECDH). Liu et al [37] have presented an efficient implementation of ECDH key exchange for MSP430 devices. tsk is mainly used to build a secure channel, and $\{M\}_{tsk}$ denotes that a message M is encrypted with tsk. All the

exchanged messages are encrypted with tsk by using a symmetric encryption algorithm. Specifically, the key features of the protocol can be summarized as follows:

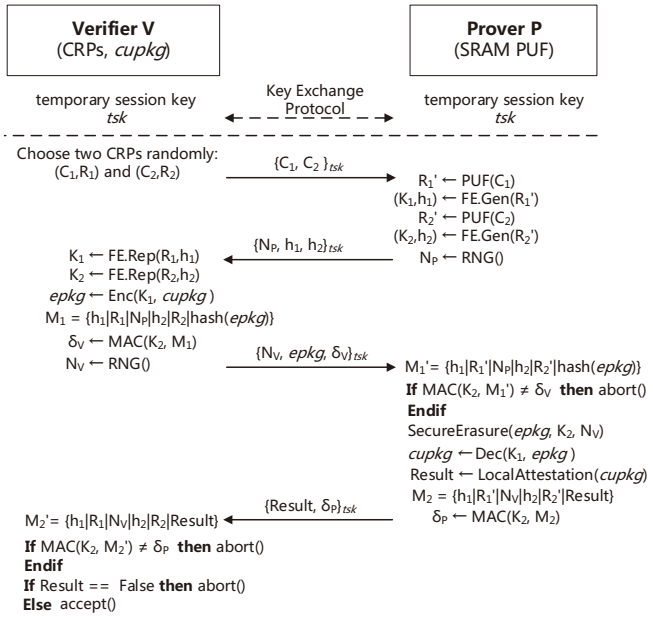

Fig. 3. Secure Code Update Protocol based on PUF

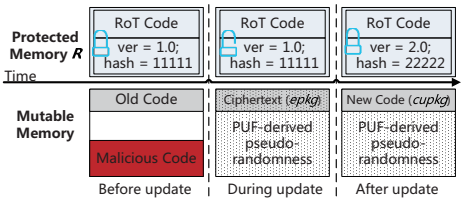

Fig. 4. Illustration of Prover's Memory Layout during Protocol Execution

(1) Key Generation based on PUF and Reverse Fuzzy Extractor without Relying on Secure Storage. V randomly chooses two CRPs (C_1, R_1) and (C_2, R_2), and sends the challenges to P. After receiving C_1 and C_2, P reads the physical PUF responses $R_1' \leftarrow PUF(C_1)$ and $R_2' \leftarrow PUF(C_2)$, and generates the secret key and helper data as $(K_1, h_1) \leftarrow FE.Gen(R_1')$, $(K_2, h_2) \leftarrow FE.Gen(R_2')$. The helper data h_1 and h_2 are sent to V, and V

uses them to recover $K_1 \leftarrow FE.Rep(R_1, h_1)$ and $K_2 \leftarrow FE.Rep(R_2, h_2)$. In this way, P doesn't need to store keys with the help of NVM-based secure storage, and can generate random keys on demand every time the protocol is started.

(2) Mutual Authentication based on K_2 and MAC. Based on the reproducibility property of PUF, V and P share the same keys K_1 and K_2 now. We use K_2 and MAC to achieve authentication. As the correct CRPs are only known to the trusted V and the physical PUF embedded in P is unclonable and unpredictable, no other party (e.g., $\mathcal{A}$) can forge a valid key. Thus, the authentication can be mutual. In detail, P generates a random nonce N_P and sends it to V. Once V receives N_P, it uses K_2 to create an authenticated message $\delta_V \leftarrow MAC(K_2, M_1)$ where M_1 contains the nonce N_P and other exchanged messages between V and P. δ_V serves as a signature, and prevents any modifications to the exchanged messages since P checks $MAC(K_2, M_1')? = \delta_V$. Similarly, δ_P is an authenticated message created by P, and verified by V.

(3) Encryption Transmission and Secure Code Erasure based on PUF. The code update package *cupkg* is encrypted by using K_1 and symmetric cryptography. Only P with a valid K_1 can decrypt the encrypted package. After P receives *epkg*, it first checks the authenticated message. If δ_V passes the verification, P believes that the messages come from an authorized V. Then P performs a secure code erasure (Algorithm 1): the encrypted package *epkg* is used to overwrite the memory occupied by the old code, and the extra memory space is filled with PUF-derived pseudorandom noises. The parameters K_2 and N_V assure that the secure code erasure is device-specific and protocol-specific, and no attackers can predict a valid memory layout. The use of *cnt* (inspired by [51]) is convenient for V to reconstruct the prover's memory layout and compute expected integrity values in advance. Secure code erasure can also eliminate possible malicious codes and restore P to a clean environment.

Algorithm 1: SecureErasure(*epkg*,K_2,N_V).

Variables:
The counter value, cnt;
The extra memory range, $[Mem_{Start} : Mem_{End}]$.

```
1 Mem(OldCode, size) ← epkg ;
2 cnt = 0;
3 for i = Mem_Start; i < Mem_End; i++ do
4     prandom ← Hash(K_2, N_V, cnt);
5     Mem[i] ← prandom;
6     cnt++;
7 end
```

(4) Local Code Integrity Attestation. After a secure code erasure, P can decrypt *epkg* and finish the installation of the update binary code. In order to attest an untampered and up-to-date software state, the RoT code in the protected region $\mathcal{R}$ triggers a local attestation routine. As illustrated in Algorithm 2, the attestation routine uses *cupkg* to perform three checks: (1) check whether the version number contained in *cupkg* is higher than the version number stored

in $\mathcal{R}$, (2) check whether the hash values in *cupkg* are different from the values stored in $\mathcal{R}$, and (3) check whether the hash values over all memory regions match the expected integrity reference values specified in *cupkg.hash* (denoted by $CheckCodeIntegrity()$). If all checks pass, the verification of code update and software integrity is successful. Upon a successful verification, RoT disables the protection on $\mathcal{R}$ and writes the newest integrity reference values (*cupkg.ver* and *cupkg.hash*) into $\mathcal{R}$. As the prover device has just performed secure code erasure and integrity attestation, no malicious values can be written into $\mathcal{R}$ at this moment. Once $\mathcal{R}$ is updated, RoT enables the write protection immediately.

Algorithm 2: LocalAttestation(*cupkg*).

```
if (cupkg.ver ≤ R.ver) ∨ (cupkg.hash == R.hash) ∨ ¬CheckCodeIntegrity(cupkg.hash)
then
    return False ;
else
    Disable protection on R ;
    UpdateR(cupkg.ver, cupkg.hash) ;
    Enable protection on R ;
    return True ;
end
```

(5) Verification Result Feedback and Secure Call. The result of local integrity attestation is included in the computation of δ_P to ensure integrity, and it is sent back to V along with δ_P. If δ_P is verified successfully, V can ensure that the result comes from the correct P as no attackers can forge K_2. According to the feedback result, V knows whether P is in an up-to-date and unmodified software state. After a successful update, RoT code in P will enforce a strict white list policy to ensure a secure code call: the entry point of the update binary application code is hardcoded in $\mathcal{R}$, and each time the control flow is passed to the application code only when $CheckCodeIntegrity()$ returns $True$.

Since SRAM is used for PUF in implementation (Sect. 4), a reboot is needed for each update protocol. In the experiment, we turn off the power manually to implement a full power cycle to collect SRAM PUF data. The initial SRAM values are used as R_1 and R_2 for each reboot, and RoT uses these values to generate K_1 and K_2. RoT is always executed after device reset, and the whole update process is handled by RoT. After the update, the keys are immediately erased by RoT. RoT also decides if the application code can be executed. Thus, we define a standard secure call sequence for P in Figure 5.

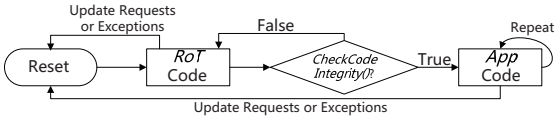

Fig. 5. Secure Call Sequence of P

Memory Integrity Check. If P's memory space is relatively large, we can divide it into multiple small sections and use hash tree (or Merkle tree) [19] to implement memory integrity check ($CheckCodeIntegrity()$). Figure 6 illustrates an example of a binary hash tree, in a setting where the memory is divided into four sections, denoted by S_1, S_2, S_3 and S_4. The hash values of these sections are the leaves, and a parent is the hash of the concatenation of its children. Only h_{root} (the root of the tree) is stored in the protected region $\mathcal{R}$. Before each update, V must decide the size of each section, and prepare a bran-new hash tree as the integrity reference value. During the update, all the sections should be checked, i.e., RoT should compute:

$Hash(Hash(Hash(S_1), Hash(S_2)), Hash(Hash(S_3), Hash(S_4)))$.

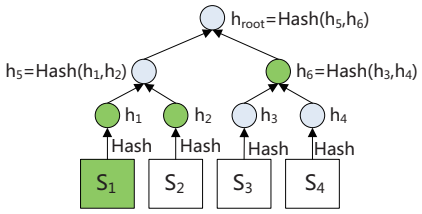

Fig. 6. A binary hash tree. Hash values of each memory section are aggregated to the root of the tree.

All intermediate values during the computation should match the hash values in the tree (including all nodes). After a succssful check, the root value h_{root} is written into $\mathcal{R}$, and other hash values are stored in the mutable memory along with the code. As the check of the entire memory is time-consuming, the hash tree method also supports to check the integrity of a specific memory section (e.g., the memory section containing the application code). For example, to check the integrity of S_1, RoT only needs to read S_1, h_2 and h_6, and the resultant aggregation value $Hash(Hash(Hash(S_1), h_2), h_6)$ is compared to h_{root}.

3.3 Analysis

The security of reverse fuzzy extractor is described in [23]. In this section, we mainly focus on the analysis of security threats (Sect. 2). We also give some comparisons and discussions about our method.

Our mechanism can defend against all mentioned security threats, and the specific analysis is as follows:

(1) **Code Alteration**. For each update, a local attestation is used to check the code integrity and any changes to the update binary code will be found.
(2) **Code Reverse Engineering**. It is almost impossible to absolutely guarantee the code confidentiality. Our main goal is to prevent code extraction during the network transmission and increase the difficulty of breaking the

prover device P. As shown in Figure 3, the communication channel only discloses *epkg*, which is encrypted with PUF-based key K_1. As we assume PUF is secure, $\mathcal{A}$ cannot decrypt *epkg*. Moreover, secure code erasure can remove any malicious code in P during the update, and at other times, RoT maintains a secure code call by enforcing a strict white list policy. Thus, it's difficult for $\mathcal{A}$ to break P and extract the update code. It should be noted that runtime attacks, control-flow attacks and physical attacks are not considered in this paper.

(3) **Loading Unauthorized Code**. The update binary code is included in *epkg*, which is sent to P along with δ_V. *epkg* is created based on K_1 and δ_V is generated based on K_2. Since K_1 and K_2 originate from the PUF of the same prover device P, it can be inferred that *epkg* is from an authorized V if δ_V is verified successfully by P. If an unauthorized *epkg* (created randomly or using a malicious key) arrives at P, its decryption is meaningless and cannot pass the verification of a local integrity attestation.

(4) **Loading Code onto an Unauthorized Device**. Due to the uniqueness and unpredictability of PUF, an unauthorized device cannot derive a correct decryption key K_1 and thus cannot install a update code intended for another device.

(5) **Code Downgrading**. An ascending version number *cupkg.ver* is included in each code update package *cupkg*, the attestation routine will check the version number.

(6) **Incomplete Update**. Firstly, the result of *LocalAttestation* is included in δ_P, and thus V can ensure the integrity and authenticity of the feedback result. Secondly, RoT resides in the protected region $\mathcal{R}$ which is write-protected and execution-uninterruptible, the only entry to RoT is reset, and the only chance to write $\mathcal{R}$ is after a secure code erasure and a $CheckCodeIntegrity()$. Since the feedback result and δ_P are created by RoT, the result $True$ indeed indicates a complete update and the result $False$ illustrates the other situations.

(7) **Alleviating TOCTOU**. It's difficult to completely prevent TOCTOU, e.g., runtime attacks may break our system easily, which are not discussed here. Our mechanism uses the post-update defense to alleviate the TOCTOU attack, which is not considered in previous update mechanisms. During each update, the code is checked in the local integrity attestation routine and the newest reference values are written to $\mathcal{R}$. After update (post-update defense), RoT checks the integrity of application code by using the newest reference values to run $CheckCodeIntegrity()$ each time the application code is called. If the code has been tampered with, RoT will never give the system control to the code. In this case, RoT could trigger a new update protocol.

(8) **Device Cloning**. Even if $\mathcal{A}$ obtains all the memory contents (including RoT code) of an authorized prover device, it cannot copy or clone a similar device to pretend to be a legitimate P because $\mathcal{A}$ cannot clone a physical PUF or predict the responses of a particular PUF.

(9) **Control-flow Attack**. Our system provides no control flow integrity, and we assume RoT code is immune from vulnerabilities. But the application

code may be compromised, we need to prevent application code from jumping to the RoT code arbitrarily. We can achieve this by enforcing a single well-defined entry point to RoT code in the ARMv8-M architecture [52]. Or in other devices, we can use software fault isolation [45] to sandbox the application code.

(10) **Physical Adversary**. Due to the unclonability and unpredictability of PUF, a physical clone or replacement of a valid prover device will be found. However, we cannot defend against other physical attacks, such as reprogramming the whole flash memory, data remanence of SRAM, or invasive attacks with micro-probing. Possible solutions to mitigate physical adversary contain the heartbeat protocol in DARPA [27].

Comparison with PoSE [42] and [35]. Our comparison with recently proposed update mechanisms mainly covers five aspects: the dependent assumptions, the supported security attributes, the ability to resist all mentioned security threats, the main communication and computation costs. As shown in Table 1, our mechanism has the following advantages: (1) Don't rely on NVM-based secure storage and a secure communication channel; (2) Resist all 8 security threats by providing security attributes like mutual authentication, confidentiality (or secrecy), integrity, unclonability and secure call; (3) The message transmitted from V to P is the size of the update binary code, and the extra memory of P is filled with PUF-derived pseudorandom numbers; and (4) Use symmetric cryptography and MAC instead of public-key cryptography and signature, which is more suitable for low-end embedded systems.

Table 1. Comparison.

	Our mechanism	[42] (ESORICS 10)	[35] (ESORICS 16)
Assumptions	Immutable Code, uninterruptible execution and a robust and unpredictable PUF	Immutable code and secure communication (P only communicates with V and no other party)	Immutable code, secure storage and uninterruptible execution
Security Attributes Supported	Freshness, Integrity, Authenticity, Confidentiality, Feasibility, Verifiability, Restorability and Secure Call	Integrity, Feasibility and Restorability	Freshness, Integrity, Authenticity, Feasibility, Verifiability, and Restorability
Resisting Security Threats	Threat-1,2,3,4,5,6,7,8	Only Threat-1	Threat-1,3,5,6
Communication costs	the size of *cupkg*	the size of P's writable memory	the size of *cupkg*
Computation costs	Symmetric cryptography, MAC, Hash, RFE	Symmetric cryptography, MAC, Hash	Symmetric and Asymmetric cryptography, Signature and verification, Hash

Comparison with Remote Attestation. Remote attestation mechanisms are mainly used for verifying the software integrity of a remote device. Our update mechanism not only verifies the integrity of a remote device after an update installation, but also needs to ensure the correctness, freshness, confidentiality and authenticity of code update. Schulz et al. [44] gave a lightweight remote

attestation by combing software-based attestation and PUF. PUFatt [36] implemented Schulz's idea by presenting a novel PUF design (called ALU PUF) based on the delay difference in two different arithmetic and logic units (ALUs). These works mainly focused on remote attestation, and did not consider secure code update. Furthermore, ALU PUF needs to change the microprocessor of device and is not available in current embedded devices. Researches (like SMART [18], Sancus [40], TyTAN [13], etc.) all tried to propose lightweight secure architecture for embedded devices, which can be used to implement remote attestation (also called hybrid attestation by [1]). In our opinion, these architecture can be easily extended to implement secure code update although none of them mentioned this. However, all hybrid attestation schemes need some hardware modifications, which are not available commercially. Our secure code update mechanism can be applicable using existed resources in current commodity embedded devices.

Limitation. Firstly, our method requires that the prover device must have enough SRAM space, meeting the memory requirements for PUF and program variables at the same time. For low-end embedded devices, we may consider increasing the size of SRAM memory or exploring new PUF primitives (like Flash-based PUF [50]). Secondly, the scalability of our scheme is not good. To update multiple devices in a large network, V has to establish an update protocol for each individual device. Even if all devices have the same configuration (that is, the same $cupkg$), V must prepare different hash reference values and different encryption package $epkg$ for different devices. Our future work will be focused on the design and implementation of a scalable and lightweight secure code update mechanism based on PUF. A preliminary idea is to combine PUF physical properties with attribute-based encryption (ABE) [2], where PUF responses can be viewed as specific attributes associated with a decryption key.

Discussion. Helfmeier et al. [22] used a Focused Ion Beam (FIB) circuit edit (CE) to successfully produce a physical clone of a SRAM PUF. Although we assume a 'good' PUF in the adversary model, it's better to strengthen SRAM PUF with synthesized logic as recommended in [22] or adopt other PUF instances (like Flash-based PUF [50]) for high-security applications. Recently, data remanence attack [4] brought a new threat to SRAM PUF, but the attack needs a harsh condition (low-temperature between -110°C and -40°C). Verifying the temperature using the sensors within embedded devices before each update may mitigate this attack. Note that, our work is not to design an ideal PUF, but to use PUF to design a secure code update mechanism. Actually, any PUF instances can be used in our update protocol. In addition, we adopt SRAM PUF because SRAM is ubiquitous in various computing devices and there are no modeling attacks currently found against weak PUFs. But we have to assume $\mathcal{A}$ cannot access the challenge-response interface of the PUF and cannot obtain temporary data stored in volatile memory during the update protocol. Although this is a strong assumption (the assumption is also used in other literatures like [23, 10]), it is necessary because no secure execution environment (like TEE) exists in current embedded devices. However, this assumption can be improved by forcing memory access control based on a Memory Protection Unit (MPU) [34,

13] or using other techniques such as obfuscation and white-box cryptography. We adopt reverse FE due to less performance overhead, actually any FEs (like a computationally secure FE [15]) can be used if they are more effective. Finally, our work mainly focuses on providing security without changing hardware for legacy devices. However, in many embedded scenes, modifying hardware is necessary to provide strong security, and we think ARM TrustZone technology in ARMv8-M architecture will be a good choice.

4 Implementation and Performance Considerations

Setup. We implement and evaluate our proposed secure code update scheme on a MSP-EXP430G2 LaunchPad Development Board. The board is a complete USB-based development and experimenter tool from Texas Instrument with a MSP430G2553 MCU by default. The key features of the MSP430G2553 MCU include [29]: ultralow-power, von-Neumann architecture; 16-bit RISC CPU (up to 16MHz); 16KB of programmable Flash; 512 bytes of SRAM.

We use the on-board SRAM as the source of entropy to implement the *PUF* and random number generator (*RNG*). For reverse fuzzy extractor (*RFE*), we adopt the BCH error correction code to eliminate noises and use a hash function as an entropy accumulator to generate unpredictable random keys. We implement the hash function using SHA256, while the symmetric algorithm uses 128-bit AES. The MAC computation is implemented by using the construct of HMAC-SHA256. As no hardware acceleration is supported in MSP430G2553, all of the cryptographic algorithms are implemented in software based on [28]. As 512B SRAM is relatively small, our implementation is based on the following guidelines: (1) Use more constants and Flash space; (2) Use fewer variables and RAM space; (3) Initial SRAM values are written to Flash used for *PUF* and *RNG*, and the actual SRAM space is reserved for global and local variables (.bss, .data and .stack). Our time performance is measured in clock cycles. As we set the clock frequency to 1MHz, m cycles are equal to $m/1,000,000$ seconds. Our evaluation code (in python) and data for PUF are uploaded to the Github[2].

SRAM PUF and SRAM RNG. We collect the startup SRAM values from two different MSP430G2553 devices, each measured over 50 power cycles. Based on these data, we first evaluate the robustness, uniqueness and randomness of SRAM PUF by analyzing the min-entropy and Hamming distance. For robustness, we compute the intra-chip Hamming distance (HD_{Intra}) between repeated measurements of SRAM cells from the same chip. The resulting HD_{Intra} is 260 (260/4096=6.3%) at average, and 743 (743/4096=18%) at worst. For uniqueness, we compute the inter-chip Hamming distance (HD_{Inter}) and min-entropy over the measurements from different chips. The average ratio for HD_{Inter} is 42.3%, and the min-entropy rate is 87% which means the average min-entropy per bit is 0.87. For randomness, we compute the min-entropy over 50 repeatedly measured SRAM values from the same chip, which gives an average min-entropy

[2] `https://github.com/vonwaist/PUFRNG`

rate of 7.76%. This means that we need at least 1/7.76%=12.88 SRAM cells to obtain one random bit. These evaluated results show a well-featured PUF.

4096-bit (=512B) SRAM space is allocated as follows: 2628 bits are used to generate two PUF CRPs, and the remaining 1468 bits are used to derive random numbers. The address spaces are separated to avoid direct correlation between *PUF* and *RNG*. As only two CRPs can be used in each device, C_1 and C_2 needs not to be transmitted over the network. Using multiple CRPs corresponds to storing multiple session keys. It means that we have two default session keys. Additionally, we use 256 bits SRAM to derive a 16-bit random nonce, which is achieved by XORing adjacent bytes 16 times. Thus, 5 (1468/256) random numbers can be used for each power cycle. Aysu et al. [10] showed that the SRAM data can pass all experiments in the NIST statistical Test Suite after 8-fold XORing, thus our 16-fold XORing is random enough. *RNG* is implemented in assembly by using only two registers (one for the start address of SRAM RNG and the other for the *xor* result). The code size of RNG is 56 bytes and it takes 44 clock cycles to output one random number. Theoretically, a random extractor should be used instead to generate RNG, we choose XOR due to low overhead and Aysu's experience in [10].

Reverse Fuzzy Extractor. A BCH($n, k, d = 2t + 1$) [39] code allows to correct errors up to t-bit within a n-bit block. We customize a BCH(127,15,53) based on the open source code[3], which can correct up to 20.5% noisy bits (greater than the worst SRAM noise level of 18%). As the average min-entropy rate for uniqueness is 87%, 1314 (2628/2) bits SRAM data contains 1143 (1314×0.87) bits entropy. We use 1143 bits PUF entropy in 9 blocks of a BCH(127,15,53) code, and 1008(=(127-15)×9) bits are leaked in the helper data. The remaining entropy is 135 (=1143-1008) bits, which are enough for a 128-bit key. We use SHA256 to hash the PUF response, and the 256-bit result is 2-XORed to obtain a 128-bit key. We assume that a single bit flips with a probability of $P_{error} = 7\%$ (greater than the average HD_{Intra}), then the probability that 27 bits or more will flip in a 127-bit block is $P_{block} = \sum_{i=27}^{n=127} \binom{127}{i} P_{error}^{i} (1 - P_{error})^{(127-i)} \approx 1.87 \times 10^{-7}$, and thus the error cannot be corrected in this case. For 9 blocks of a BCH(127,15,53) code, the probability that a key can be fully reconstructed is $P_{correct} = (1 - P_{block})^9 > 1 - 1.69 \times 10^{-6}$.

The *PUF* and *RFE.Gen* are implemented in C with a code size of 3274 bytes, and it also uses 768 bytes constant space and 426 bytes variable space. To save RAM, we pre-compute the coefficients of the generator polynomial, log table and antilog table of the Galois field GF(2^m), and store these parameters as the constants in the flash memory. The implementation contains four steps: read SRAM values to generate a 1314-bit PUF response (it takes 1471 cycles); use SHA256 and 2-XORing to generate a 128-bit key (it takes 290,951 clock cycles); BCH Encoder for 9 blocks (it takes 585,504 clock cycles); write the result to Flash (132,982 cycles).

Symmetric Algorithm, Hash and MAC. There is a decryption operation for each update protocol, and we adopt 128-bit AES algorithm. The code size of

[3] `http://www.eccpage.com/`

Dec is 910 bytes, and the memory requirements for its constants and variables are 522 bytes and 119 bytes, respectively. To decrypt a 128-bit cipher text, *Dec* takes about 23,487 CPU cycles. The hash function is SHA256, and its implementation costs 1530 bytes of code size, 288 bytes of constant space and 271 bytes of variable space. The performance of SHA256 depends on the specific input size, e.g., 96,617 cycles for 50-byte input, 291,040 cycles for 150-byte. HMAC is implemented based on SHA256, and its code size is 2348 bytes. For a 16-byte message, HMAC-SHA256 takes about 392,174 clock cycles. As many MCUs support cryptographic hardware security[4], the performance can be improved further.

Secure Erasure and Local Attestation. Two algorithms $SecureErasure()$ and $LocalAttestation()$ are both implemented based on SHA256. The code size of $SecureErasure()$ is 2,568 bytes, and the number of clock cycles it takes to erase a 512B flash section is 1,615,880. The main time consumption of $SecureErasure()$ is caused by SHA256 computation and Flash write operation. The primary role of $LocalAttestation$ is $CheckCodeIntegrity()$, which is also the most time-consuming part. $CheckCodeIntegrity()$ computes the hash value of a given memory block and compares it with the reference value, and it takes about 292,422 clock cycles for a 128-byte application program code.

Protected Memory. In our method, write-protection is needed for storing the version, reference hashes and RoT code, and we use existing hardware resources in embedded devices to implement a static non-volatile write-protected region $\mathcal{R}$. In MSP430G2553, the hardware resources are Flash memory. According to [29], the Flash memory of MSP430G2553 is partitioned into main and information memory sections. The information memory has four 64-byte segments, and the main memory has multiple 512-byte segments. The information memory can be locked separately from the main memory with a LOCKA bit. When LOCKA is set, the information memory is protected and cannot be written or erased. Thus, RoT code can be stored in the information memory. As the size (256-byte) of information memory in MSP430G2553 is smaller than the size of our RoT code, our evaluation described above uses the main memory. However, this does not affect the evaluation results because there are no other differences between the information and main memory except for the lock bit.

In MSP430FR family [30], the protected hardware resources are FRAMs similar for MPU. An FRAM is a non-volatile memory that can be read and written like a standard SRAM. An MPU can be used to divide the device's main memory into three variable-sized segments with configurable read, write and execute access. Furthermore, the protection of the second segment can be temporarily removed when necessary by the bootloader, which can be used to store and update the integrity reference hash values. Bootloader (similar to our RoT) locks the MPU settings before jumping to the application, preventing the application from corrupting or overwriting the protected area. For the security of PUF, we propose to allow only the RoT code to access the start-up values at boot time, and after that the SRAM space is erased by RoT.

[4] `http://www.ti.com/ww/en/embedded/security/index.shtml`

For uninterruptible execution, we suggest to disable the interrupt during the execution of RoT code. Before the control is handed over to the applicaton code, RoT enables the interrupt and at the same time checks the integrity of application code and all interrupt handlers.

Comparison with Public-key Cryptography. As the MSP430G2553 device does not have enough resources to implement and run a ECC/RSA algorithm, we compare our PUF-based AES encryption (with 128-bits key) with a RSA encryption (with 2048-bits key) in a host environment. RSA is implemented based on the open-source mbed TLS library[5]. For a 100-bytes plain message, we test the two encryption operations 1000 times respectively. The min, max and average runtime for PUF-based AES encryption are 0.023ms, 1.927ms, and 0.0549ms; and the runtime for RSA are 1.076ms, 16.07ms, and 1.37ms. Obviously, our method is more lightweight and more suitable for tiny embedded devices. In the future, we plan to purchase a more rich embedded development board (e.g., MSP430FR family) to make a more comprehensive comparison.

5 Related Work

Remote Attestation. Remote attestation can be categorized in three main branches: hardware-based attestation, software-based attestation and hardware-software co-design with minimum hardware requirements. Hardware-based attestation relies on strong hardware features, such as TCG's TPM [41, 8], ARM TrustZone [6] and Intel SGX [5], which are not supported on low-cost commodity embedded devices. Software-based attestation [46, 48, 7, 26] does not require secure hardware and thus is well suitable for constrained embedded systems. However, its security guarantee is weak. Between the two mechanisms, hardware-software co-design [18, 40, 34, 36, 13] aims to build a dynamic trust anchor in a low-end embedded device with minimal changes to existing MCUs. The trust anchor established can be further used to design a scalable collective attestation protocol (SEDA [9] and SANA [3]), meeting the global security requirements of large groups of interconnected smart devices. In our opinion, all remote attestation mechanisms can be used to strengthen secure updates, e.g., to verify the code integrity after update. But a complete secure code update is more than a remote attestation mechanism.

Secure Code Updates For Embedded Devices. SCUBA [47] is a secure code update mechanism by using software-based attestation to ensure indisputable code execution (ICE) on a remote sensor node. PoSE [42] is a different approach that can enable a prover device to convince a verifier that it has erased all its memory. As the overhead of PoSE is relatively high, some researchers try to explore effective skills to reduce the overhead including uncomputable hash function [17], invert-hash PoSE and graph-based PoSE [33], and All or Nothing Transforms [32]. Recently, Kohnhauser and Katzenbeisser [35] presented a novel code update scheme which verifies and enforces the correct installation of code

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

updates on all commodity low-end embedded devices in a mesh network. To address the security threats involved with the in-field firmware updates process, Texas Instruments [31, 30, 21] proposes to integrate cryptographic algorithms and security mechanisms into the bootloader of its ultra-low-energy MCUs.

SRAM PUF. The SRAM PUF was first introduced in 2007 by Holcomb et al. [24, 25] and Guajardo et al. [20] concurrently and independently. Holcomb et al. [24, 25] proposed to use SRAM physical fingerprints for identification and generation of true random numbers in RFID tag circuits, while Guajardo et al. [20] used initial SRAM values to design new protocols for IP protection on FPGAs. To provide a viable alternative to costly protected non-volatile memory (NVM), Maes et al. [38] presented a low-overhead implementation of helper data algorithm for SRAM PUFs using soft decision information. The SRAM PUF was implemented and evaluated on a microcontroller in [12]. Researchers from intrinsic-ID showed the construction of a FIPS 140-3 compliant random bit generator based on SRAM PUF in [49], and presented a comparative analysis of several types of SRAM memories from different technology nodes and demonstrated the reliability and uniqueness of all the tested SRAMs when used as PUFs in [43]. Aysu et al. showed in [10] that SRAM PUF can be used to design and implement a provably secure protocol that supports privacy-preserving mutual authentication.

6 Conclusion

In this paper, we presented a novel secure code update scheme for commodity low-end embedded devices by combing the advantages of secure erasure and physically unclonable function. We concluded eight security threats that may happen in secure code updates from the existing literature, and showed how our scheme can be used to prevent or mitigate these threats. Our scheme doesn't rely on secure storage or secure communication. By using the symmetric cryptography and lightweight construction of a reverse fuzzy extractor, our approach offers acceptable communication and computation overhead. Finally, we also eliminate the gap from the world of protocol theory to concrete realization through evaluating all protocol components in a single TI MSP430 device. Our implementation uses only on-board SRAM and the protected memory resources without requiring any hardware modifications, which is applicable to a broad range of popular low-end embedded systems.

7 Acknowledgments

The work has been supported by the National Natural Science Foundation of China (No.61602455 and No.61402455). We thank anonymous reviewers for their helpful comments. We specially thank Aurlien Francillon for his suggestions on improving our paper.

References

1. T. Abera, N. Asokan, L. Davi, F. Koushanfar, A. Paverd, A. R. Sadeghi, and G. Tsudik. Invited: Things, trouble, trust: On building trust in iot systems. In *53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.
2. M. Ambrosin, A. Anzanpour, M. Conti, T. Dargahi, S. R. Moosavi, A. M. Rahmani, and P. Liljeberg. On the feasibility of attribute-based encryption on internet of things devices. *IEEE Micro*, 36(6):25–35, Nov 2016.
3. Moreno Ambrosin, Mauro Conti, Ahmad Ibrahim, Gregory Neven, Ahmad-Reza Sadeghi, and Matthias Schunter. Sana: Secure and scalable aggregate network attestation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 731–742, NY, USA, 2016. ACM.
4. Nikolaos Athanasios Anagnostopoulos, Stefan Katzenbeisser, Markus Rosenstihl, Andr Schaller, Sebastian Gabmeyer, and Tolga Arul. Low-temperature data remanence attacks against intrinsic sram pufs. Cryptology ePrint Archive, Report 2016/769, 2016. `http://eprint.iacr.org/2016/769`.
5. Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R 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.
6. ARM. Arm security technology: Building a secure system using trustzone technology. Technical report, ARM Technical White Paper, 2009.
7. Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. A security framework for the analysis and design of software attestation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 1–12, New York, NY, USA, 2013. ACM.
8. Will Arthur and David Challener. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. Apress, Berkely, CA, USA, 1st edition, 2015.
9. N. Asokan, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter, Gene Tsudik, and Christian Wachsmann. Seda: Scalable embedded device attestation. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 964–975, NY, USA, 2015. ACM.
10. Aydin Aysu, Ege Gulcan, Daisuke Moriyama, Patrick Schaumont, and Moti Yung. End-to-end design of a puf-based privacy preserving authentication protocol. In *Cryptographic Hardware and Embedded Systems*, CHES 2015, pages 556–576, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
11. Christoph Bhm and Maximilian Hofer. *Physical Unclonable Functions in Theory and Practice*. Springer Publishing Company, Incorporated, 2012.
12. C. Bohm, M. Hofer, and W. Pribyl. A microcontroller sram-puf. In *Network and System Security (NSS), 2011 5th International Conference on*, pages 269–273, Sept 2011.
13. Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. Tytan: Tiny trust anchor for tiny devices. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 34:1–34:6, New York, NY, USA, 2015. ACM.
14. Sergey Bratus, Nihal D'Cunha, Evan Sparks, and Sean W. Smith. Toctou, traps, and trusted computing. In *Proceedings of the 1st International Conference on Trusted Computing and Trust in Information Technologies: Trusted Computing*

- *Challenges and Applications*, Trust '08, pages 14–32, Berlin, Heidelberg, 2008. Springer-Verlag.
15. Ran Canetti, Benjamin Fuller, Omer Paneth, Leonid Reyzin, and Adam Smith. Reusable fuzzy extractors for low-entropy distributions. In *Proceedings of the 35th Annual International Conference on Advances in Cryptology — EUROCRYPT 2016 - Volume 9665*, pages 117–146, New York, NY, USA, 2016. Springer-Verlag New York, Inc.
16. Yevgeniy Dodis, Rafail Ostrovsky, Leonid Reyzin, and Adam Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. *SIAM J. Comput.*, 38(1):97–139, March 2008.
17. Stefan Dziembowski, Tomasz Kazana, and Daniel Wichs. One-time computable self-erasing functions. In *Proceedings of the 8th Conference on Theory of Cryptography*, TCC'11, pages 125–143, Berlin, Heidelberg, 2011. Springer-Verlag.
18. Karim Eldefrawy, Aurélien Francillon, Daniele Perito, and Gene Tsudik. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *NDSS 2012, 19th Annual Network and Distributed System Security Symposium, February 5-8, San Diego, USA*, San Diego, UNITED STATES, 02 2012.
19. Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Caches and hash trees for efficient memory integrity verification. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, pages 295–, Washington, DC, USA, 2003. IEEE Computer Society.
20. Jorge Guajardo, Sandeep S. Kumar, Geert-Jan Schrijen, and Pim Tuyls. Fpga intrinsic pufs and their use for ip protection. In *Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '07, pages 63–80, Berlin, Heidelberg, 2007. Springer-Verlag.
21. Oscar Guillen, Bhargavi Nisarga, Luis Reynoso, and Ralf Brederlow. Cryptobootloader secure in-field firmware updates for ultra-low power mcus, texas instruments incorporated, 2015.
22. C. Helfmeier, C. Boit, D. Nedospasov, and J. P. Seifert. Cloning physically unclonable functions. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 1–6, June 2013.
23. Anthony Van Herrewege, Stefan Katzenbeisser, Roel Maes, Roel Peeters, Ahmad-Reza Sadeghi, Ingrid Verbauwhede, and Christian Wachsmann. Reverse fuzzy extractors: Enabling lightweight mutual authentication for puf-enabled rfids. In *Financial Cryptography*, volume 7397 of *Lecture Notes in Computer Science*, pages 374–389. Springer, 2012.
24. Daniel E Holcomb, Wayne P Burleson, and Kevin Fu. Initial sram state as a fingerprint and source of true random numbers for rfid tags. In *Proceedings of the Conference on RFID Security*, volume 7, 2007.
25. Daniel E Holcomb, Wayne P Burleson, and Kevin Fu. Power-up sram state as an identifying fingerprint and source of true random numbers. *IEEE Transactions on Computers*, 58(9):1198–1210, 2009.
26. Julian Horsch, Sascha Wessel, Frederic Stumpf, and Claudia Eckert. Sobtra: a software-based trust anchor for arm cortex application processors. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 273–280. ACM, 2014.
27. Ahmad Ibrahim, Ahmad-Reza Sadeghi, Gene Tsudik, and Shaza Zeitouni. Darpa: Device attestation resilient to physical attacks. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, WiSec '16, pages 171–182, New York, NY, USA, 2016. ACM.

28. Texas Instruments Incorporated. C implementation of cryptographic algorithms, slaa547a-july 2013, 2013.
29. Texas Instruments Incorporated. Msp430x2xx family user's guide, slau144j-december 2004, revised july 2013, 2013.
30. Texas Instruments Incorporated. Crypto-bootloader (cryptobsl) for msp430fr59xx and msp430fr69xx mcus, user's guide, slau657-november 2015, 2015.
31. Texas Instruments Incorporated. Secure in-field firmware updates for msp mcus, application report, slaa682-november 2015, 2015.
32. Ghassan O. Karame and Wenting Li. Secure erasure and code update in legacy sensors. In *Proceedings of the 8th International Conference on Trust and Trustworthy Computing*, TRUST'15, pages 283–299, Cham, 2015. Springer International Publishing.
33. Nikolaos P. Karvelas and Aggelos Kiayias. Efficient proofs of secure erasure. In *Proceedings of Security and Cryptography for Networks: 9th International Conference*, SCN '14, pages 520–537, Cham, 2014. Springer International Publishing.
34. Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 10:1–10:14, New York, NY, USA, 2014. ACM.
35. Florian Kohnhauser and Stefan Katzenbeisser. Secure code updates for mesh networked commodity low-end embedded devices. In *The 21st European Symposium on Research in Computer Security*, ESORICS '16, pages 320–338. Springer, 2016.
36. Joonho Kong, Farinaz Koushanfar, Praveen K. Pendyala, Ahmad-Reza Sadeghi, and Christian Wachsmann. Pufatt: Embedded platform attestation based on novel processor-based pufs. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 109:1–109:6, NY, USA, 2014. ACM.
37. Zhe Liu, Hwajeong Seo, Zhi Hu, Xinyi Hunag, and Johann Grosschadl. Efficient implementation of ecdh key exchange for msp430-based wireless sensor networks. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 145–153, New York, NY, USA, 2015. ACM.
38. Roel Maes, Pim Tuyls, and Ingrid Verbauwhede. Low-overhead implementation of a soft decision helper data algorithm for sram pufs. In *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '09, pages 332–347, Berlin, Heidelberg, 2009. Springer-Verlag.
39. Roel Maes, Anthony Van Herrewege, and Ingrid Verbauwhede. Pufky: A fully functional puf-based cryptographic key generator. In *Proceedings of the 14th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'12, pages 302–319, Berlin, Heidelberg, 2012. Springer-Verlag.
40. Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 479–494, Berkeley, CA, USA, 2013. USENIX Association.
41. B. Parno, J. M. McCune, and A. Perrig. Bootstrapping trust in commodity computers. In *2010 IEEE Symposium on Security and Privacy*, SP '10, pages 414–429. IEEE Computer Society, May 2010.
42. Daniele Perito and Gene Tsudik. Secure code update for embedded devices via proofs of secure erasure. In *Proceedings of the 15th European Conference on*

Research in Computer Security, ESORICS'10, pages 643–662, Berlin, Heidelberg, 2010. Springer-Verlag.
43. Geert-Jan Schrijen and Vincent van der Leest. Comparative analysis of sram memories used as puf primitives. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pages 1319–1324, San Jose, CA, USA, 2012. EDA Consortium.
44. Steffen Schulz, Ahmad-Reza Sadeghi, and Christian Wachsmann. Short paper: Lightweight remote attestation using physical functions. In *Proceedings of the Fourth ACM Conference on Wireless Network Security*, WiSec '11, pages 109–114, New York, NY, USA, 2011. ACM.
45. David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary cpu architectures. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.
46. A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. Swatt: software-based attestation for embedded devices. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 272–282, May 2004.
47. Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Scuba: Secure code update by attestation in sensor networks. In *Proceedings of the 5th ACM Workshop on Wireless Security*, WiSe '06, pages 85–94, NY, USA, 2006. ACM.
48. Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 1–16, NY, USA, 2005. ACM.
49. Vincent van der Leest, Erik van der Sluis, Geert-Jan Schrijen, Pim Tuyls, and Helena Handschuh. *Efficient Implementation of True Random Number Generator Based on SRAM PUFs*, pages 300–318. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
50. Yinglei Wang, Wing-kei Yu, Shuo Wu, Greg Malysa, G. Edward Suh, and Edwin C. Kan. Flash memory for ubiquitous hardware security functions: True random number generation and device fingerprints. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 33–47, Washington, DC, USA, 2012. IEEE Computer Society.
51. Yi Yang, Xinran Wang, Sencun Zhu, and Guohong Cao. Distributed software-based attestation for node compromise detection in sensor networks. In *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, pages 219–230, Oct 2007.
52. Joseph Yiu. White paper: Armv8-m architecture technical overview, 2015.
53. Shijun Zhao, Qianying Zhang, Guangyao Hu, Yu Qin, and Dengguo Feng. Providing root of trust for arm trustzone using on-chip sram. In *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices*, TrustED '14, pages 25–36, New York, NY, USA, 2014. ACM.