ASSURED: Architecture for Secure Software Update of Realistic Embedded Devices

N. Asokan *Fellow, IEEE*, Thomas Nyman, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik *Fellow, IEEE*

Abstract—Secure firmware update is an important stage in the IoT device life-cycle. Prior techniques, designed for other computational settings, are not readily suitable for IoT devices, since they do not consider idiosyncrasies of a realistic large-scale IoT deployment. This motivates our design of ASSURED, a secure and scalable update framework for IoT. ASSURED includes all stakeholders in a typical IoT update ecosystem, while providing end-to-end security between manufacturers and devices. To demonstrate its feasibility and practicality, ASSURED is instantiated and experimentally evaluated on two commodity hardware platforms. Results show that ASSURED is considerably faster than current update mechanisms in realistic settings.

Index Terms—Computer Security, Embedded Software, Internet of Things, Embedded Systems

1 Introduction

Deploying insecure Internet-of-Things (IoT) devices can have disastrous consequences, as demonstrated by large-scale IoT botnets, such as Mirai [2] and Reaper[1]. IoT devices are ideal malware targets, for several reasons: First, Internet-connected devices are inherently more exposed to remote exploitation. Second, embedded systems are notoriously difficult to update, which often leaves known vulnerabilities unpatched. Third, many such devices operate in a mostly unattended fashion, which means that timely discovery of compromise is unlikely.

Once an IoT device is deployed, the ability to remotely update its device's firmware is critical to maintaining security over its lifetime. In many real-world scenarios, devices must be deployed in remote or inaccessible locations, rendering physical (manual) maintenance impossible or prohibitively expensive. Remote *"Over-the-Air"* (OTA) delivery of firmware updates allows manufacturers to deliver new features or functionality, as well as to patch bugs and flaws. However, if designed poorly, insecure update mechanisms may be exploited by the adversary, causing victim devices to malfunction, cease operation, or fall under adversarial control.

Some prior update techniques (geared for different settings) meet security requirements under specific assumptions. For example, TUF [31] is an update delivery framework resilient to key compromise, while its descendant Uptane [22] extends and adapts TUF to support secure updates for automotive systems. However, both techniques requires direct interaction between the manufacturer and the devices in order to specify device-specific constraints on the update process. This makes them unsuitable for large-scale IoT deployments, where updates may be delivered via broadcast, or from third-party *Content Delivery Networks* (CDNs). Hence, the update mechanism can not rely on interactive protocols or on transport-level security. Also, TUF and Uptane do not support verification of proper update installation on target devices.

Some proposals for secure firmware updates on resource-constrained devices (e.g., SCUBA [32] and PoSE [30]) allow the updater to obtain a verifiable proof of successful update. However, they involve strong assumptions (e.g., an optimal checksum function or strictly local communication), or lack support for update robustness, i.e., roll-back to a previous firmware version if the current update fails. Such issues make them unsuitable for realistic IoT deployments. This is discussed further in Section 2 which overviews related work.

Goals and Contributions:

- *IoT Update Ecosystem:* We identify essential roles in the IoT secure software update ecosystem and show that they cannot be directly incorporated into state-of-the-art secure update methods [31], [22]. We also identify objectives for an IoT secure software update system. (Section 3)
- *Secure Firmware Update Framework:* We propose ASSURED, which: (a) provides end-to-end security by combining an existing reliable update delivery framework (e.g., TUF) with an authorization mechanism that allows manufacturers to specify update constraints, and (b) allows a local authority to specify constraints for – and verify – successful update deployment. (Section 4)
- *Realization & Evaluation:* We instantiate ASSURED on two low-end security architectures: HYDRA [16] and ARM TrustZone-M [4]. We also demonstrate its practicality via proof-of-concept implementations on two commodity platforms: I.MX6-SabreLite [8] (Section 5.1) and ARM Cortex-M23 microcontroller prototyping system

- *Author names are listed in alphabetical order.*
- *N. Asokan and T. Nyman are with the Secure Systems Group of Aalto University, Espoo 02150, Finland. Portions of this research were done while T. Nyman was with Trustonic Oy, Helsinki, Finland. E-mail: asokan@acm.org, thomas.nyman@aalto.fi.*
- *N. Rattanavipanon and G. Tsudik are with the Computer Science Department, University of California, Irvine, CA 92697-3435, USA. E-mail: {nrattana,gts}@uci.edu.*
- *A-R. Sadeghi is with the Technische Universität Darmstadt, Darmstadt 64289, Germany. E-mail: ahmad.sadeghi@trust.tu-darmstadt.de.*

1. https://www.arbornetworks.com/blog/asert/reaper-madness/

This article was presented in the International Conference on Embedded Software 2018 and appears as part of the ESWEEK-TCAD special issue.

equipped with TrustZone-M (Section 5.2). Our evaluation shows that ASSURED improves upon current update architectures in terms of deployability and performance in realistic IoT settings. It also meets the objectives we identified for different stakeholders in the IoT update ecosystem.

2 Background & Prior Work

This section overviews several related topics.

2.1 Boot Integrity

Platform boot integrity is a fundamental requirement for any system designed to resist copying, corruption, or compromise. *Secure* or *authenticated boot* [29] mechanisms examine integrity of the system's software components at boot time, thus detecting changes to the system's trusted state.

In *secure boot*, each step in the boot process verifies a public key signature on the next step in the boot chain, before it is launched. The source of trust in the secure boot process typically originates from a *Static Root-of-Trust*, such as an immutable piece of code and a private key, imprinted (hard-coded) by the device manufacturer. A software image must be signed by its manufacturer before deployment, making it impractical to verify configuration information provided by the system administrator that controls the device during its operation.

In *authenticated boot*, each step of the boot process is measured, e.g., by computing a cryptographic hash over the software image and platform configuration information; the resulting measurement is stored in a way that allows it to be securely retrieved later. Unlike secure boot, authenticated boot permits any software component to run. However, the securely stored measurement can be used for local access control decisions (e.g., access to hardware-based keys), or for producing a signed statement of the system's state to a remote verifier, as described in Section 2.2. Authenticated boot relies on a Root-of-Trust for guaranteeing unforgeability of measurements.

Secure or authenticated boot are standard features in modern PC [28] and mobile platforms [5], although their architectural realizations can differ significantly across platforms. Boot integrity is also important for embedded platforms, where its use has been mainly to protect against memory corruption [13]. For instance, virtually all microcontroller units (MCUs) check operating integrity at initialization, or during recovery from a low-voltage condition, e.g., by computing a *Cyclic Redundancy Checksum* (CRC) of the software image, and comparing it with a CRC stored in persistent storage, typically flash memory. However, CRC-based checks are do not defend against attacks on device's boot integrity, since an attacker who modifies the code on the device can bypass the CRC check via specially crafted software images, or even by modifying the reference CRC in flash. Therefore, modern MCU platforms employ cryptographic hash algorithms (instead of CRCs) and one-time-programmable fuses to store reference measurements used for secure boot. The use of cryptographic algorithms for secure boot and for communication in resource-constrained MCUs triggers inclusion of cryptographic hardware accelerators, even in very small MCUs.

2.2 Remote Attestation

Remote attestation is a process whereby a trusted entity (verifier) remotely measures internal state of a untrusted and possible compromised device (prover), in order to determine whether the latter is in a benign state. Current remote attestation approaches can be partitioned into three groups: hardware-based, software-based and hybrid. Hardware-based attestation relies on security provided by dedicated hardware features such as a Trusted Platform Module (TPM) [36] or Intel's SGX [12]. Such hardware features are generally not viable for resource constrained IoT devices, such as MCUs, due to their complexity and cost.

On the other hand, software-based attestation requires no hardware features at all. Instead, it assumes: (1) consistent timing characteristics of the measurement process on the prover, (2) existence of an optimal (space- and time-wise) checksum function [33], [26], [34]. Unfortunately, these assumptions only hold when attestation is performed over one-hop communication, along with an idealized checksum function. Consequently, software-based methods are unsuitable for remote attestation in realistic settings, e.g., over the Internet.

Hybrid remote attestation is exemplified by SMART [17] architecture. It imposes minimal changes to existing MCUs. SMART requires immutability of attestation code and key by storing them in a read-only memory region. SMART also utilizes hardwired MCU access control rules to ensure that: (1) access to the attestation key is restricted to attestation code, and (2) execution of attestation code is atomic, i.e., uninterruptible and executed as a whole. A follow-on result, TrustLite [25], provides a more flexible way to specify these access control rules. Access control configuration in TrustLite can be programmed in software at compile time and enforced by an additional feature, EA-MPU: Execution-Aware Memory Protection Unit. Unlike SMART, TrustLite does not require uninterruptible execution of attestation code, since its CPU Exception Engine is modified to support secure interrupt handling. A subsequent result, TyTan [9], extends TrustLite to support dynamic access control configuration and real-time guarantees.

2.3 Secure Updates

The Update Framework (TUF)[31] is a generic security framework designed to integrate with existing software repositories. TUF adds a new layer of signed metadata, including file sizes and cryptographic hashes of file content. Figure 1 (① through ③) illustrates the sequence of events in TUF-based update distribution. TUF clients (end-hosts) periodically poll repositories for changes and fetch this metadata (①), and new software artifacts as needed (②). By verifying the metadata (③), clients detect whether files or metadata have been manipulated.

TUF assigns responsibility of signing different parts of metadata to different *roles*. In order to improve resilience against key compromise, all roles can use one or more distinct key-pairs and require clients to validate a threshold number of signatures of the role's keys. TUF defines four fundamental roles necessary to meet its security goals: *root*, *targets*, *snapshots*, and *timestamp*.

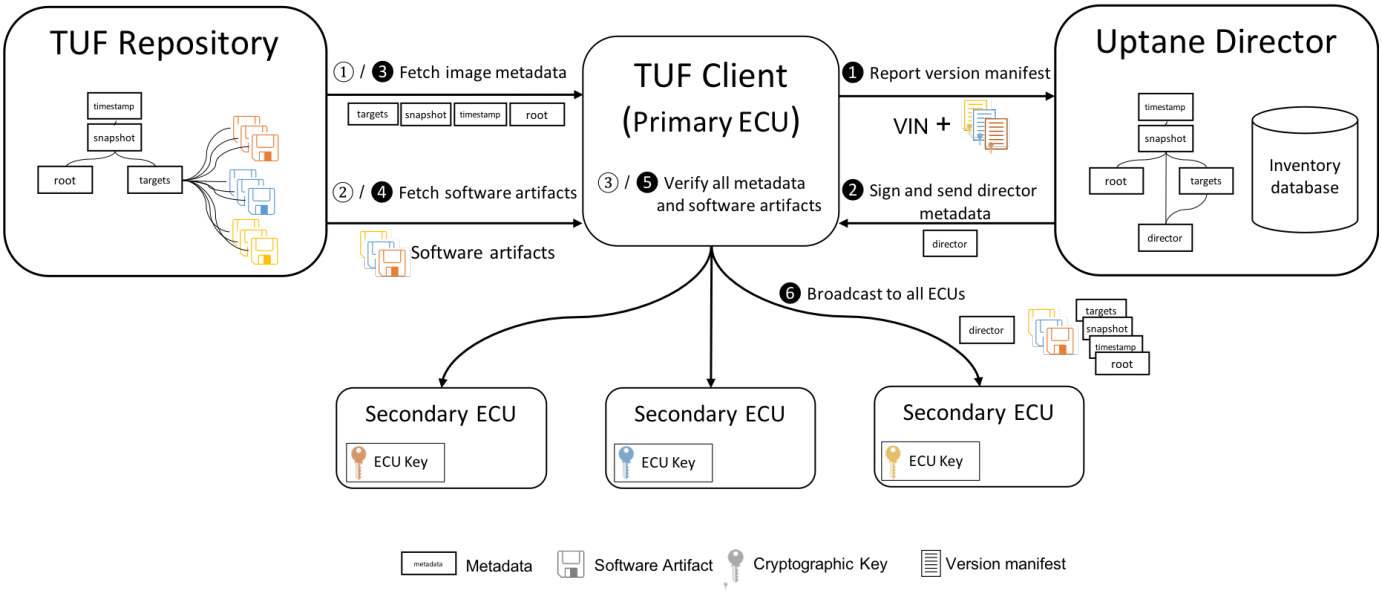

Fig. 1: Sequence of update distribution events in TUF and Uptane. ① to ③ depict the update process for TUF, while ❶ to ❻ depict the extended update process employed by Uptane.

The root role acts as a Certification Authority (CA) for the repository. It signs public keys of all other top-level roles. TUF clients must receive the root role's trusted public keys out-of-band, e.g., at manufacture or install time. Since the root role's keys act as roots-of-trust in TUF, they should be stored offline, physically disconnected from the Internet to minimize the risk of compromise.

The targets role signs metadata describing software artifacts which can be trusted by clients. Since software may be originated by different sources, the targets role may delegate full or partial trust to an auxiliary role with a separate set of key-pairs. Partial delegation limits the set of files that the role is allowed to indicate as trusted. A role with delegated trust can delegate this ability further.

The snapshots role signs metadata that confirms the latest version of all other TUF metadata stored in the repository, except the timestamp role metadata described below.

All TUF metadata is associated with an expiration time. In addition, *the timestamp role* periodically signs a statement indicating the latest version of the snapshot metadata even if there have been no updates.

Uptane [22] is an instantiation of TUF customized for software distribution to automotive systems. In order to manage updates to numerous and diverse computerized components found in modern vehicles, Uptane extends TUF with an additional *director* repository. This repository allows an *Original Equipment Manufacturer* (OEM) more control of software images deployed in individual ECUs.

Figure 1 (❶ through ❻) illustrates the sequence of events in Uptane-based update distribution. Uptane designates one Engine Control Unit (ECU) of each vehicle as *primary*; it orchestrates delivery of updates between the repository and *secondary* ECUs. As part of the update process, the primary ECU reports a *vehicle version manifest* (containing a signed statement from each ECU about its software configuration) to the remote director repository, along with the *Vehicle Identification Number* (VIN) (❶). The director repository determines the correct, up-to-date software configuration for each ECU in the vehicle identified by the VIN. It also signs director metadata that contains instructions bound to the unique serial number of each ECU, describing all software artifacts each ECU must install (❷). The primary ECU fetches and verifies all metadata and software artifacts on behalf of secondary ECUs (❸ through ❺) and distributes them over the vehicle's local-area network (❻). The metadata is broadcast to all ECUs.

To protect against compromise of the primary ECU or man-in-the-middle attacks (MiTM) attack originating within the vehicle's internal network, each secondary ECU reverifies the metadata and software artifacts it receives from the primary ECU. However, each secondary ECU might not be equipped to fully verify or store all repository metadata. To accommodate such ECUs, Uptane relaxes verification requirements for *partial verification* ECUs, which only receive and verify director's metadata and software artifacts specified therein for that ECU.

2.4 Secure Update via Remote Attestation

Several prior secure update techniques use pre-existing remote attestation designs. For example, *SCUBA* [32] can be used to repair a compromised sensor through firmware updates. SCUBA utilizes an authentication mechanism and software-based attestation to identify memory regions infected by malware and transmits the repair update to replace these regions. However, the attestation technique based on self-checksumming code heavily relies on consistent timing characteristics of the measurement process the use of an optimal checksum function. Due to these assumptions, SCUBA is not a suitable approach for IoT settings [11].

Perito and Tsudik [30] present a simple secure firmware update technique using so-called Proofs of Secure Erasure (PoSE-s). The idea is for prover to perform secure erasure

before retrieving and installing a new update, from clean slate. Secure erasure is achieved by filling all of prover's memory with (uncompressable) randomness chosen by verifier. Prover then returns a snapshot of the new memory contents to verifier as a proof of secure erasure. This proof guarantees that prover is now in a benign clean state and ready to perform an update. Subsequent work [21], [23] improved the original method by reducing time, energy and bandwidth overheads. However, all PoSE-based update techniques work only in a one-hop prover/verifier setting. Furthermore, they do not support *robust updates*, i.e. ability to retain previous version(s), in case a roll-back is required, e.g., if the current update cannot be successfully completed. Whereas, in ASSURED, prover can isolate untrusted software, thus secure erasure is not needed and update robustness can be achieved.

2.5 ARM TrustZone

ARM microprocessors are RISC-based low-power processors that can be found in many modern devices, such as smartphones, smart TVs, smart watches, tablets and various other computing devices. *Cortex-A* series application processors are commonly deployed in mobile devices, networking equipment and other home and consumer devices. *Cortex-M* series embedded processors are used in MCUs that require low cost and energy efficiency, such as sensors, wearables and small robotic devices.

Since 2003, ARM application processors have featured *TrustZone* Security Extensions [3], a hardware feature that aims to reduce attack surface of security-critical code by separating processor operation into two distinct states: *Secure* and *Non-secure*. In Non-secure state, hardware-based access control prevents less trusted software (OS and applications) from accessing resources belonging to the Secure state. The ARMv8-M architecture also supports TrustZone Security Extensions in Cortex-M cores. Guarantees provided by TrustZone in Cortex-M processor are similar to that by TrustZone in Cortex-A [4], although the microarchitectural realizations of TrustZone Security Extensions differ significantly between Cortex-A and Cortex-M processors.

ARM-based IoT devices commonly utilize either low-end Cortex-A processors ($<$ 1GHz cores typically *without* TrustZone) or Cortex-M microcontrollers, ranging from few tens to a few hundred MHz. The latest additions to the Cortex-M processor family: Cortex-M23[2] and Cortex-M33[3], also feature TrustZone Security Extensions.

3 System Model

In this section, we identify essential stakeholders in the IoT firmware update ecosystem. We then specify anticipated adversarial capabilities and assumptions. Next, we describe the requirements for ASSURED and discuss how to realize them on a low-end device.

3.1 Stakeholders

We adopt the same stakeholder model as the one used in the Software Updates for Internet of Things (SUIT) Working Group[4] of the Internet Engineering Task Force (IETF) [27]. It includes four types of stakeholders:

- **Original Equipment Manufacturer (*OEM*)**. Produces devices, issues the initial firmware and releases subsequent updates. During manufacturing, *OEM* can securely install cryptographic keys on its devices [5]
- **(Software) Distributor**. Essentially plays the role of a surrogate in the update distribution process. Since *OEMs* may not wish to build and maintain the complex infrastructure to support logistics of software distribution at scale, update distribution can be outsourced to software distributors, such as CDNs.
- **(Domain) Controller**. Responsible for the upkeep, configuration, and reliable operation of devices within its administrative domain. The domain may be defined by physical proximity, e.g., devices in the vicinity that are reachable from Controller via local connectivity, e.g., WiFi or Bluetooth Low Energy. Alternatively, the domain that Controller is responsible for may be defined organizationally, e.g., in cases where Controller is operated as a cloud-hosted service.
- **(Connected) Device**. The ultimate target of updates. We focus on resource-constrained (low-end) connected devices, such as: sensors, actuators, hybrids of both, or any other embedded devices that operate under strict resource limitations in terms of memory, storage and processing power.

The *OEM* and Controller must be able to specify constraints on updates to be deployed on Device. For instance, *OEM* might use the same signing key to sign updates for different device variations. Thus, it needs to ensure that a particular device only installs updates for the correct variation. A software update might also be issued to enable or disable a feature for a particular subset of devices, e.g., enable debug for a development device. Controller may wish to constrain update deployment time, i.e., during a regular maintenance window, or when a device is otherwise idle. Updates may also be deployed as differential patches, i.e., updates do not contain full software images, but only the changes between the previous software version, and the updated version. In such cases the *OEM* must be able to place constraints on the order in which the updates are installed to ensure that each patch applies cleanly, and the resulting software configuration in Device remains consistent at all times.

3.2 Adversary Model

We base our adversarial model on the subset of realistic attack types enumerated in [1]:

I *Remote Adversary* can compromise untrusted file servers or cloud storage infrastructure components that store firmware updates before they reach Device. Remote adversary may also attempt to remotely exploit software vulnerabilities, in order to infect Device with malware.

2. https://developer.arm.com/products/processors/cortex-m/cortex-m23
3. https://developer.arm.com/products/processors/cortex-m/cortex-m33
4. https://datatracker.ietf.org/wg/suit/about/
5. While software itself may be produced by a third-party developer, we assume that OEM always controls its distribution. While not inconceivable, we are unaware of any cases of a software developer directly distributing firmware and/or its updates to IoT devices from multiple OEMs.

II *Local Adversary* is sufficiently near Device to intercept communication and generally interfere with network traffic between Distributor and Controller or device-to-device communication between Controller and Device.

As in prior related literature, we consider attacks on Device-s by so-called *physical adversaries* to be out-of-scope. However, we note that physical attacks can be mitigated via tamper-resistant techniques, or using communication-intensive (though unscalable) absence detection [20].

3.3 Objectives

We identify several objectives for a secure and reliable update framework applicable to realistic IoT devices. These objectives also have some overlaps with the existing firmware update requirements discussed in [27].

O1 *End-to-End Security:* Device must verify that a firmware update it receives is originated by OEM, and OEM must specify device-specific constraints on the update. However, due to large numbers of devices, OEM may not be able to directly interact with all.

O2 *Update Authorization from Controller:* Controller must control which firmware updates must be installed on Device. This implies that Device must verify whether firmware updates are approved by Controller for installation.

O3 *Attestation of Update Installation:* Controller must obtain a verifiable proof of successful update installation on Device.

O4 *Protection of Code & Secret Keys on Device:* ensure confidentiality and integrity of code and secret keys used in update and attestation processes.

O5 *Minimal Burden for Device:* impose minimal computational and storage burden on Device.

TUF and Uptane do not satisfy all of these requirements in realistic IoT scenarios. In particular, TUF also requires Device itself to make policy decisions about which updates to fetch and install, violating O5. In addition, several security features of TUF require multiple signature verifications using different keys, which makes TUF computationally expensive, further violating O5.

Uptane overcomes these issues by introducing the director repository to provide update decisions for each device and limits verification requirements for resource-constrained devices to only verifying director signatures. However, since the director repository is held by OEM, Uptane implies direct interaction between OEM and Device, which violates O1. TUF and Uptane do not consider the client device as part of their threat models and simply assumes overall security of the device, not satisfying O4. Lastly, neither of them specifies the need for an external entity to validate correct update installation, which violates O3.

3.4 Device Prerequisites

To meet aforementioned objectives, Device's security architecture must include at least the following:

- **Secure or Authenticated Boot:** to guarantee authenticity and integrity of trusted software at boot time. This generally requires a minimal hardware root-of-trust, e.g., as in [17], [25].
- **Isolated Execution:** to protect trusted security-critical operations on Device from being influenced by untrusted (potentially vulnerable or malicious) code.
- **Secure Storage:** to ensure that trust anchors used for firmware update validation and attestation are integrity-protected and only accessible by authorized trusted software at run-time.

These requirements can be satisfied by modern embedded device platforms that support either (1) TrustZone Security Extensions [3] or (2) a secure microkernel, e.g., seL4 [24]. Section 5 discusses instantiations of our secure update framework on these two architectures.

4 Design

Our goal is to extend any update distribution scheme to allow OEM and Controller to specify constraints on the update process. As an example, we extend TUF with ASSURED and show how Device can use ASSURED constraints to decide whether to install updates it receives. ASSURED can be combined seamlessly with TUF on Controller to benefit from TUF's security guarantees. As a result, besides security of TUF and Uptane, ASSURED satisfies additional OEM requirements on update distribution. We discuss how ASSURED satisfies the objectives in Section 6.

ASSURED expects Device to implement the necessary mechanisms to meet O1, O2, and O3. However, Device is not expected to perform full verification of TUF metadata. In Section 5, we show that, as a result, ASSURED compares favorably to TUF in terms of computational and storage burdens on Device. It is thus very suitable for IoT deployments involving resource-constrained devices.

4.1 Sequence of Events

OEM prepares *Software Artifacts* for distribution by emitting cryptographic authorizations that can be verified by Device to determine if software artifacts are sanctioned by OEM. An *authorization token* encodes constraints in the form of metadata that is recognized by Device, e.g., device model or a unique device identifier. This metadata must be validated by Device when deciding if the software artifact should be installed. An authorization token must always include a signature computed with the OEM's authorization key on the hash of the constraints and the software artifact itself.

Figure 2 shows the sequence of events during update distribution and delivery. OEM emits an authorization token and encapsulates authorization information together with the corresponding software artifact in an *Update Envelope* (❶). OEM uploads the resulting envelope and its metadata to the *TUF Repository* (❷) where the envelope is recorded into the repository's TUF metadata. The TUF Repository is then mirrored by an untrusted Distributor.

Controller, acting as a TUF client on behalf of Device, periodically polls the repository for updates. When new update envelopes appear, Controller fetches the snapshot and targets metadata, validates them and fetches any new envelopes intended for Device (❸). At this point, each envelope is validated against the corresponding record in the targets metadata (❹). Controller can now arbitrate on local update policies that may apply to the fetched software

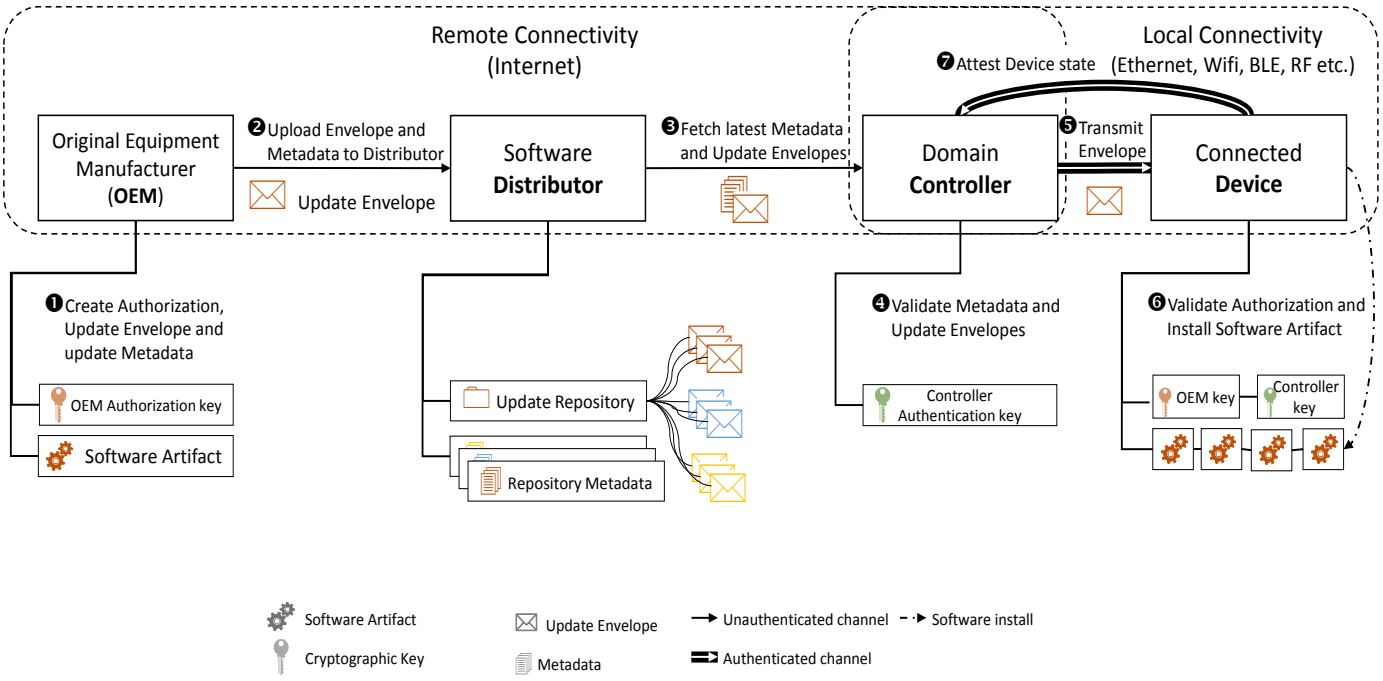

Fig. 2: Sequence of events during update distribution and delivery.

artifact. It then transmits the update envelope (that it decides should be installed) to Device over an authenticated channel (❺). This channel serves as an implicit authorization from Controller that it has approved the software artifact in the transmitted update envelope. Device uses its underlying security architecture to securely validate authenticity and integrity of the OEM's authorization token and software artifact in the update envelope. If the signature and constraints are valid, it installs the software artifact (❻). We note that the security architecture of Device guarantees the protection of code and secret keys on Device. Thus, O4 is achieved in this step. Finally, Controller attests the state of Device to ensure that the software artifact is successfully installed (❼). The last step allows Controller to obtain a verifiable proof when the update process is complete, which satisfies O3. Meanwhile, if the update process fails (e.g., by the adversarial preventing an update from reaching Device), Controller will be able to detect it due to the incorrect or missing response.

4.2 Authorization Mechanism

The mechanism for authorizing software updates must satisfy O1 and O2. Namely, it must allow Device to authenticate the source of software updates as well as let OEM and Controller specify applicable constraints. In ASSURED, we identify two concrete approaches for realizing authorization tokens that meet both needs:

Extension of TUF Targets Metadata. OEM can create an authorization token for each software artifact and embed it into the TUF targets metadata. This allows OEM to define update constraints for specific software artifact, as well as allows Controller to validate update metadata of different devices separately. As a result, metadata associated with software artifact SA in the targets metadata can now be encoded as:

$$Auth_{SA} := [hash(SA), size(SA), C, Sig(K_{OEM}, hash(SA)||size(SA)||C)]$$

where C denotes constraints, e.g., device model and/or unique device identifier.

Adoption of GP TMF. Alternatively, an authorization token can be delivered to Device encapsulated in the Update Envelope, using GlobalPlatform TEE Management Framework (TMF) [18]. TMF is a security model for administration of TEEs. GlobalPlatform-compliant TEEs based on ARM TrustZone [3] are widely deployed, especially on Android devices. TMF defines the set of administration operations available to various parties in the administration of a TEE and its Trusted Applications (TAs). TMF also defines a security model that allows business relationships and responsibilities to be mapped to a set of *Security Domains*, and a security layer for the authentication and establishment of secure communication channels between such parties and corresponding security domains.

The subset of TMF needed to support ASSURED authorization is only the *Update TA* command [18, Section 8.4.3] and the use of TMF's explicit authorization [18, Section 5.2.1] primitives. Explicit authorization allows TMF commands to be authenticated when there is no means of establishing a direct communication channel between the party that signs the authorization, and the on-device security domain acting on behalf of that party, such as in the case of broadcast channels and update repositories. In the context of our framework, OEM signs a TMF Authorization Token with associated constraints (such as the applicable device model) and emits a TMF Envelope that encapsulates the Software Artifact, Update TA command, and TMF Authorization Token.

Both approaches could be realized in either TrustZone-M and HYDRA architectures. However, since TrustZone-M is likely to be found on low-end MCUs, implementations of ASSURED based on concise binary encoding of update

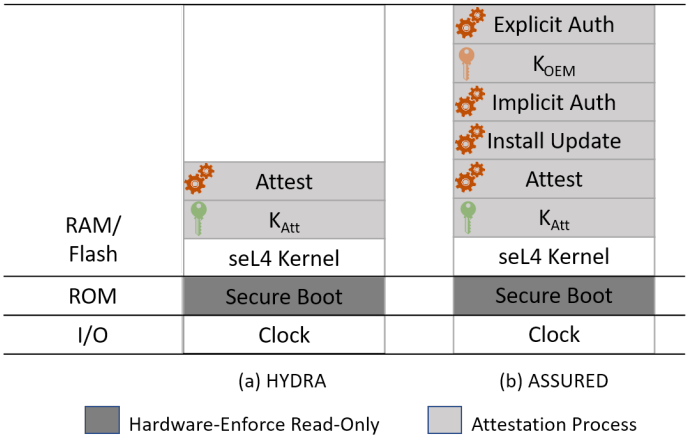

Fig. 3: Memory organization of HYDRA-based firmware update.

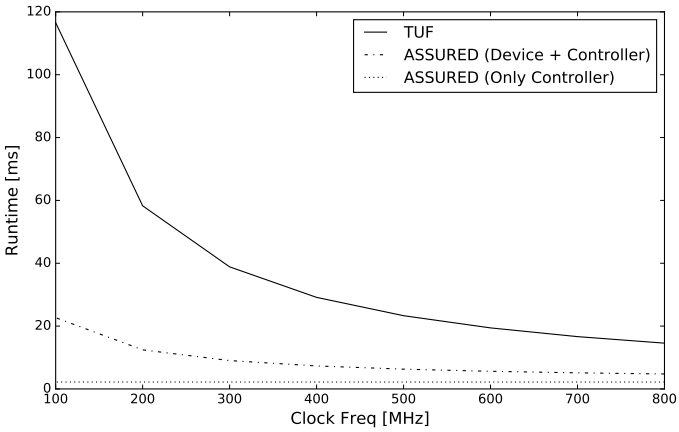

Fig. 4: Total runtime of TUF and ASSURED for variable Device clock frequencies (I.MX6-SabreLite). Controller clock frequency is fixed at 3.4GHz.

metadata, such as *Concise Binary Object Representation* [7] or *Abstract Syntax Notation One* (ASN.1) are more suitable for TrustZone-M devices compared to TUF JSON objects. TMF is based on a subset of the ASN.1 Distinguished Encoding Rules, and in addition provides an existing set of constraints that can be easily extended [18, Section 5.3.2]. Alternatively, OEM or Controller can encode $Auth_{SA}$ in fixed-size formats suitable for parsing on severely restricted devices.

5 Implementation

In this section we describe two proof-of-concept implementations of ASSURED.

5.1 ASSURED on HYDRA

We now overview HYDRA, discuss implementation details and report on experimental evaluation.

5.1.1 HYDRA Overview

HYDRA implements a hybrid (HW/SW) remote attestation design by building upon the formally verified seL4 [24] microkernel, which provably guarantees process memory isolation and enforces access control to memory regions. Using the formally proven isolation features of seL4, access control rules can be implemented in software and enforced by the microkernel. Figure 3a summarizes memory organization of HYDRA. HYDRA implements secure storage for the attestation key (K_{Att}) by storing it in a writable memory region and configuring the system, such that no other process, besides the attestation process (PR_{Att}), can access this memory region. Access control configuration in HYDRA guarantees strong isolated execution of PR_{Att} by enforcing exclusive access to its thread control block as well as to its memory regions. To ensure uninterruptibility, HYDRA runs PR_{Att} as the so-called *initial user-space process* with the highest scheduling priority. As the initial user-space process in seL4, PR_{Att} is initialized with capabilities that allow access to all available resources.

Meanwhile, the rest of user-space processes are assigned lower priorities and their resource access is limited by PR_{Att}. HYDRA also requires a *reliable read-only clock* to defend against denial-of-service attacks via replayed, delayed or reordered attestation requests [10]. Finally, hardware-enforced secure boot feature is used to ensure integrity of seL4 itself and of the initial process when the system is initialized.

5.1.2 Implementation Details

Figure 3b shows the implementation of ASSURED as part of PR_{Att} in HYDRA. Specifically, we modify PR_{Att} to support the TMF-style authentication mechanism via implicit and explicit authorization operations.

Following TMF specifications [19], we use AES [35] and HMAC-SHA256 [14] as the underlying cryptographic primitives to ensure implicit authorization from Controller via a secure channel. In particular, PR_{Att} derives encryption and MAC keys (used during the setup of the secure channel) from a pre-shared master K_{Att}. Once established, the secure channel between PR_{Att} and Controller yields new session keys used to protect the transmitted update envelope.

For explicit authorization, we assume OEM's authorization key (K_{OEM}) is distributed and pre-installed on devices out-of-band, e.g., during manufacturing. To ensure confidentiality and integrity, PR_{Att} protects K_{OEM} the same way as K_{Att}. Elliptic Curve-based signature scheme, ED25519 [6][6], is ported to seL4 and serves as the underlying signature scheme to provide explicit authorization operation in PR_{Att}.

5.1.3 Evaluation

We compare performance of ASSURED and TUF in terms of code size and runtime on a popular commercially available platform: I.MX6-SabreLite [8]. We chose TUF as a point of comparison since it (and its variants) is currently the only established secure update standard relevant to IoT [15].

Code & Metadata Size. As shown in Table 2, ASSURED adds around $6.7K$ lines of C code to HYDRA's code-base, while overall size of PR_{Att} executable increases by 9KB. About 67% of code overhead is due to ED25519 code. In order to minimize runtime from parsing metadata, we

6. ED25519 is chosen because it is shown to run faster than other existing signature schemes while still providing the same security guarantees.

	TUF	ASSURED		
		Expl. Auth.	Impl. Auth.	Total
Verification Time (ms)	14.57	2.46	0.1	2.56
Metadata Size (bytes)	940	136	52	188

TABLE 1: Performance comparison between ASSURED and TUF on I.MX-SabreLite @ 800MHz. Verification of TUF metadata is performed using TUF-recommended threshold values: 2 for root and targets roles, and 1 for others. TUF metadata size is estimated assuming only one target file in the targets role.

	HYDRA		ASSURED	ED25519 Impl.
	Attestation	Net. & Libs.		
Code Size (KLOC)	12	94	2.2	4.5
Executable Size (KB)	250.6		9.4	

TABLE 2: Code and executable sizes of PR_{Att} on I.MX6-SabreLite.

encode ASSURED's metadata in a fixed-size format instead of JSON. An ASSURED update envelope carries 188 bytes of metadata, while the size of TUF metadata is estimated to be around 940 bytes. (See Table 1 for more details.)

Runtime Overhead. Table 1 shows the runtime comparison between ASSURED and TUF implemented on top of HYDRA. Recall that we use ED25519 [6] as the signature scheme for both methods. In a typical scenario, full verification of all TUF metadata takes much longer ($\sim$ 5.7 times) on Device, for two reasons. First, since TUF metadata is encoded in JSON format, parsing it on Device as part of the update process consumes a non-negligible amount of time. In our experiments, this takes around $1ms$ or $\sim$ 6% of total runtime. However, the major reason for this significant increase is because TUF full verification requires at least 6 public key operations. In contrast, ASSURED offloads these operations to Controller, and Device only performs lighter-weight computation, i.e., validating an OEM authorization token received from Controller via an authenticated channel. Figure 4 shows that Controller performs TUF full verification in $2.2ms$.

Next, we assess runtime performance of the entire ASSURED process, i.e., combined runtime of ASSURED in both Device and Controller, and compare it to runtime for Device to perform full TUF verification. Results in Figure 4 show that ASSURED is still considerably faster than TUF and the difference becomes more significant as Device's clock frequency drops. This clearly serves as a motivation to offload this computationally expensive task to Controller.

5.2 ASSURED on ARM Cortex-M23

We now describe a proof-of-concept implementation of ASSURED on a Cortex-M23 MCU and report on its experimental evaluation.

5.2.1 ARM Cortex-M23 Overview

As described in Section 2.5 ARM Cortex-M23 MCU is equipped with TrustZone Security Extensions that allow partitioning the system into *secure* (trusted) and *non-secure* (untrusted) execution environments. (Sometimes these are referred to as separate secure and non-secure *"worlds"*) separated from each other by hardware.) A context switch between them is performed by the hardware processor logic when specific conditions are met. The processor logic ensures that code in the non-trusted execution environments can enter trusted code only at specific entry points, and that non-trusted code remains strongly isolated from resources (e.g., memory and interrupt lines) belonging to the trusted execution environment.

At system boot, the MCU starts execution in the trusted execution environment. Although the boot flow might vary between specific Systems-on-Chip (SoCs), it typically begins from bootstrap code stored in secure ROM that validates and starts a trusted bootloader, e.g., based on a trust root for verification, often reflected in the hash of a code signature verification key stored in one-time-programmable fuses.

The trusted bootloader configures access control rules for memory partitioning to separate trusted code and data from their non-trusted counterparts. Secure storage can be realized by simply storing sensitive keys (or other data) in memory allocated to the trusted execution environment. The trusted bootloader can also adjust interrupt priorities and interrupt line assignments to ensure that trusted code receives priority when deciding which interrupt handler routines are invoked to service processor events.

To ensure that non-trusted code can not change device's software configuration, persistent storage used as code memory (e.g., internal flash) can be configured to be only writable by trusted code. Code re-programming support may be exposed to non-trusted code via APIs provided by trusted software. These APIs can implement authentications to decide if re-programming of code is allowed.

5.2.2 Implementation Details

We implemented ASSURED as part of the trusted bootloader on PR_{Att}. In this variant, we only support explicit authorization via a pre-configured trust root for verification in the form of a public authorization key (K_{OEM}), which is embedded into the trusted bootloader software image placed on Device during manufacturing. $Auth_{SA}$ is encoded in a fixed-size format. As in ASSURED on HYDRA, we use ED25519 as the signature scheme.

K_{Att} is established with Controller during enrollment and stored in secure memory. $Auth_{SA}$ is stored in persistent storage on PR_{Att}, and used during Device boot to validate non-trusted software artifacts (SAs). If Device has sufficient memory to store both the current software artifact SA_n

	TUF	ASSURED
Verification Time (ms)	10723	1816
Attestation Response Time (ms)	N/A	517
Metadata Size (bytes)	940	136

TABLE 3: Performance comparison between ASSURED and TUF on ARM V2M-MPS2+ configured as a Cortex-M23 @ 25MHz. Verification of TUF metadata is performed using the same parameters as in Table 1.

	Bootloader	ASSURED	ED25519 Impl.
Code Size (LoC)	959	5448	4322
Executable Size (KB)	—— 48 ——		17

TABLE 4: Code and executable sizes of PR_{Att} on ARM V2M-MPS2+.

and the next update SA_{n+1}, $Auth_{SA_{n+1}}$ is validated before SA_n is reprogrammed with SA_{n+1}. However, if Device can not store both SA_n and SA_{n+1} simultaneously, SA_n is overwritten by SA_{n+1} and only then validated. In the latter case, if SA_{n+1} validation fails, a replacement SA must be obtained from Controller. We recommend that a back-up copy of the trusted bootloader is kept when updating the trusted bootloader itself to ensure that the update process remains robust. Hence, it is important to minimize the impact of ASSURED on the trusted bootloader code size.

5.2.3 Evaluation

We assess performance – in terms of code and size and runtime – of ASSURED for resource-constrained MCUs on ARM Versatile Express Cortex-M Prototyping System MPS2+ FPGA (ARM V2M-MPS2+)[7] configured as a Cortex-M23 MCU running at 25 MHz.

Code & Metadata Sizes. Table 4 shows the impact of ASSURED on trusted bootloader code size. Most of the increase in code size is attributed to the ED25519 implementation – $\approx$ 80%. The size of $Auth_{SA}$ is a mere 136 bytes.

Runtime Overhead. To compare ASSURED with TUF, we adapted the TUF implementation from Section 5.1 to run on ARM Cortex-M23 MCU. As before, we used ED25519 as the signature scheme for both ASSURED and TUF. TUF uses a fixed-size encoding for its metadata. Table 1 shows the runtime comparison between ASSURED and TUF on Cortex-M23. Our assessment of runtime performance of ASSURED includes validation $Auth_{SA}$ and SA on Device, compared with full TUF verification. We also measured the time for Device to generate its attestation response.

The evaluation shows that ASSURED outperforms TUF (when using full metadata verification) by a factor of 4.5 in terms of total time spent for metadata verification and attestation response generation.

6 Meeting Stated Objectives

We use descriptions of ASSURED design and realization (in Sections 4 and 5, respectively) to informally argue that ASSURED satisfies all objectives stated in Section 3.3.

O1 *End-to-End Security:* ASSURED requires OEM to include an authorization token in each update envelope. End-to-end security with constraints between OEM and Device is thus guaranteed, since the token represents explicit authorization from OEM, which can be validated by Device without the need to establish a direct communication channel with OEM.

O2 *Update Authorization from Controller:* ASSURED requires Controller to transmit an update envelope to Device through an authenticated channel. This serves as implicit authorization by Controller that it has approved the software artifact contained in the envelope.

O3 *Attestation of Update Installation:* At the end of ASSURED's sequence of events, Device must reply to Controller with an attestation result that reflects its current software state. This allows Controller to determine whether the update has been correctly installed on Device.

O4 *Protection of Code & Secret Keys on Device:* The underlying HYDRA architecture provides secure storage for secret keys using capability-based access control configuration, and isolated execution of critical code guaranteed by seL4. Also, our specific hardware platform (SabreLite) provides hardware-enforced secure boot of seL4. In ARM Cortex-M23, this property is satisfied similarly by: (1) TrustZone Security Extensions that allow partitioning for a secure environment and (2) a secure boot chain anchored in ROM-resident bootstrap code. Therefore, both implementation of ASSURED (on HYDRA and ARM Cortex-M23) satisfy all Device requirements and meets this objective.

O5 *Minimal Burden for Device:* As experimental results show, ASSURED considerably lowers computational burden on Device, by off-loading heavy computational tasks to Controller. However, we do not claim that the incurred overhead is truly minimal.

7 Conclusion

This paper motivates the need for, and constructs ASSURED – a secure firmware update framework for the large-scale IoT setting with resource-constrained devices. ASSURED extends TUF – the popular state-of-the-art secure update mechanism. ASSURED takes into account realistic stakeholders in large-scale IoT deployments while providing end-to-end security with enforceable constraints between device manufacturers and IoT devices. ASSURED offloads heavy computational operations to more powerful entities and places minimal burden on IoT devices. Practicality of

7. https://www.keil.com/boards2/arm/v2m_mps2/

ASSURED is demonstrated via two prototype implementations on HYDRA and ARM TrustZone-M architectures. Experimental evaluations show that ASSURED incurs very low overhead, particularly for end-devices.

ACKNOWLEDGMENTS

Research by UCI co-authors was supported in part by: (1) DHS, under subcontract from HRL Laboratories, (2) ARO under contract W911NF-16-1-0536, and (3) NSF WiFiUS Program Award 1702911. Research by Aalto University co-authors was supported in part by (1) Academy of Finland under grant nr. 309994 (SELIoT) under the auspices of the WiFiUS program, (2) Business Finland under grant nr. 3881/31/2016 (CloSer), and (3) the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (CARS).

REFERENCES

[1] T. Abera, N. Asokan, L. Davi, F. Koushanfar, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "Things, trouble, trust: on building trust in IoT systems," in *DAC*, 2016.

[2] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis *et al.*, "Understanding the mirai botnet," in *USENIX Security*, 2017.

[3] ARM Ltd., "ARM security technology - building a secure system using TrustZone technology," http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c, 2009.

[4] ——, "TrustZone technology for the ARMv8-M architecture," http://goo.gl/RyHQ2i, 2016.

[5] N. Asokan, K. Kostiainen, E. Reshetova, A.-R. Sadeghi, L. Davi, A. Dmitrienko, and S. Heuser, *Mobile Platform Security.*, ser. Synthesis Lectures on Information Security, Privacy, & Trust. Morgan & Claypool Publishers, 2014, no. Vol. 9. [Online]. Available: http://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=688023&site=ehost-live&authtype=sso&custid=ns192260

[6] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *Journal of Cryptographic Engineering*, 2012.

[7] C. Bormann and P. Hoffman, "Concise Binary Object Representation (CBOR)," Internet Requests for Comments, RFC Editor, RFC 7049, October 2013.

[8] Boundary Devices, "BD-SL-I.MX6," 2017. [Online]. Available: https://boundarydevices.com/product/sabre-lite-imx6-sbc/

[9] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: tiny trust anchor for tiny devices," in *DAC*, 2015.

[10] F. Brasser, K. B. Rasmussen, A.-R. Sadeghi, and G. Tsudik, "Remote attestation for low-end embedded devices: the prover's perspective," in *DAC*, 2016.

[11] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the difficulty of software-based attestation of embedded devices," in *ACM CCS*, 2009.

[12] V. Costan and S. Devadas, "Intel SGX explained." *IACR Cryptology ePrint Archive*, 2016.

[13] J. Donovan, "Securing MCU designs," https://www.digikey.com/en/articles/techzone/2013/oct/securing-mcu-designs, Oct. 2013.

[14] D. Eastlake 3rd and T. Hansen, "US secure hash algorithms (SHA and HMAC-SHA)," Tech. Rep., 2006.

[15] E. Eitel, "Advanced Telematic Systems – technology for the connected car," https://advancedtelematic.com/en/press-releases/ats-is-integrating-the-uptane-security-framework-for-over-the-air-software-updates-to-connected-vehicles.html, 2017.

[16] K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "HYDRA: Hybrid design for remote attestation (using a formally verified microkernel)," in *ACM WiSec*, 2017.

[17] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and minimal architecture for (establishing dynamic) root of trust," in *NDSS*, 2012.

[18] GlobalPlatform, "GlobalPlatform Device Technology, TEE Management Framework, Version 1.0," https://www.globalplatform.org/specificationform.asp?fid=7866, 2016.

[19] ——, "GlobalPlatform Device Technology, TMF: Symmetric Cryptographic Security Layers, Version 1.0," https://www.globalplatform.org/specificationform.asp?fid=7867, 2017.

[20] A. Ibrahim, A.-R. Sadeghi, G. Tsudik, and S. Zeitouni, "DARPA: Device attestation resilient to physical attacks," in *ACM WiSec*, 2016.

[21] G. Karame and W. Li, "Secure erasure and code update in legacy sensors," in *Trust*, 2015.

[22] T. Karthik, A. Brown, S. Awwad, D. McCoy, R. Bielawski, C. Mott, S. Lauzon, A. Weimerskirch, and J. Cappos, "Uptane: Securing software updates for automobiles," in *Escar Europe*, 2016.

[23] N. Karvelas and A. Kiayias, "Efficient proofs of secure erasure," in *SCN*, 2014.

[24] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "seL4: Formal verification of an os kernel," in *ACM SIGOPS*, 2009.

[25] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," in *EuroSys*, 2014.

[26] Y. Li, J. McCune, and A. Perrig, "VIPER: verifying the integrity of peripherals' firmware," in *ACM CCS*, 2011.

[27] B. Moran, M. Meriac, and H. Tschofenig, "A firmware update architecture for internet of things devices," https://tools.ietf.org/id/draft-moran-suit-architecture-01.html, 2017.

[28] G. Paul and J. Irvine, "Take control of your PC with UEFI secure boot," *Linux J.*, vol. 2015, no. 257, Sep. 2015. [Online]. Available: http://dl.acm.org/citation.cfm?id=2846055.2846056

[29] S. Pearson, *Trusted Computing Platforms: TCPA Technology in Context*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002.

[30] D. Perito and G. Tsudik, "Secure code update for embedded devices via proofs of secure erasure." in *ESORICS*, 2010.

[31] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine, "Survivable key compromise in software update systems," in *ACM CCS*, 2010.

[32] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla, "SCUBA: Secure code update by attestation in sensor networks."

[33] 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," in *ACM SIGOPS Operating Systems Review*, 2005.

[34] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla, "Swatt: Software-based attestation for embedded devices," in *IEEE S&P*, 2004.

[35] The OpenSSL Project, "OpenSSL 1.1.0-pre7-dev," https://github.com/openssl/openssl/, 2016.

[36] Trusted Computing Group., "Trusted Platform Module (TPM)," http://www.trustedcomputinggroup.org/work-groups/trusted-platform-module, 2017.