Dominance as a New Trusted Computing Primitive for the Internet of Things

Meng Xu*†, Manuel Huber‡†, Zhichuang Sun§†, Paul England¶, Marcus Peinado¶,
Sangho Lee¶, Andrey Marochko¶, Dennis Mattoon¶, Rob Spiger‖ and Stefan Thom‖

*Georgia Institute of Technology ‡Fraunhofer AISEC §Northeastern University ¶Microsoft Research ‖Microsoft

***Abstract*—The Internet of Things (IoT) is rapidly emerging as one of the dominant computing paradigms of this decade. Applications range from in-home entertainment to large-scale industrial deployments such as controlling assembly lines and monitoring traffic. While IoT devices are in many respects similar to traditional computers, user expectations and deployment scenarios as well as cost and hardware constraints are sufficiently different to create new security challenges as well as new opportunities. This is especially true for large-scale IoT deployments in which a central entity deploys and controls a large number of IoT devices with minimal human interaction.**

Like traditional computers, IoT devices are subject to attack and compromise. Large IoT deployments consisting of many nearly identical devices are especially attractive targets. At the same time, recovery from root compromise by conventional means becomes costly and slow, even more so if the devices are dispersed over a large geographical area. In the worst case, technicians have to travel to all devices and manually recover them. Data center solutions such as the Intelligent Platform Management Interface (IPMI) which rely on separate service processors and network connections are not only not supported by existing IoT hardware, but are unlikely to be in the foreseeable future due to the cost constraints of mainstream IoT devices.

This paper presents CIDER, a system that can recover IoT devices within a short amount of time, even if attackers have taken root control of every device in a large deployment. The recovery requires minimal manual intervention. After the administrator has identified the compromise and produced an updated firmware image, he/she can instruct CIDER to force the devices to reset and to install the patched firmware on the devices. We demonstrate the universality and practicality of CIDER by implementing it on three popular IoT platforms (HummingBoard Edge, Raspberry Pi Compute Module 3 and Nucleo-L476RG) spanning the range from high to low end. Our evaluation shows that the performance overhead of CIDER is generally negligible.

I. INTRODUCTION

The Internet of Things (IoT) continues to experience rapid growth. The number of IoT devices is believed to have surpassed the number of mobile phones in 2018 and is expected to reach 18 billion devices by 2022, thus accounting for almost two thirds of all internet-connected computing devices [1]. This growth is driven by a wide range of applications including home automation (e.g., smart locks and thermostats), industrial automation, connected vehicles, smart cities, water management, agriculture [2], and even vending machines [3].

†These authors participated in this work during their internships at Microsoft Research.

In a world in which the great majority of internet-connected devices are IoT devices and in which many aspects of critical infrastructure, industrial production, and everyday life depend on such devices, the Internet of Things becomes an attractive target for attackers [4]. In addition to simple resource hijacking, as demonstrated by the Mirai and Hajime Botnets [5–7], new and more pernicious types of ransomware, targeting production facilities and critical infrastructure like the power grid [8], are likely threats. The first instances of IoT ransomware have already appeared in the wild [9–11].

At the same time, IoT devices have many of the properties that have allowed traditional computing devices to be compromised on countless occasions. There is no reason to believe that the operating systems and applications running on IoT devices will be free from vulnerabilities, especially since much of the software running on these devices (e.g., the Linux kernel) were ported from traditional devices. Configuration errors such as those exploited by Mirai are also likely to persist into the future. All these problems are exacerbated by the market dynamics of a rapidly evolving emerging technology for which time-to-market pressures have traditionally taken precedence over software assurance.

Rather than attempting to design IoT devices that cannot be exploited—a daunting task under commercial constraints—this paper aims to unconditionally recover IoT devices even after the most severe compromise of the device firmware. More precisely, our goal is to enable the owner or administrator of a large IoT deployment to install and run a firmware version of his or her choice even if the device firmware is under the control of an attacker and actively resists recovery. We call the ability of one computing device (i.e., the owner's IoT back-end server) to control the software configuration of another computing device *dominance*.

Dominance can be seen as a stronger version of attestation [12–15]. Whereas attestation allows the back-end to identify the software on a device, dominance enables the back-end to dictate it. Attestation has been a critical primitive in the design of many trusted computing systems [16–21] that protect confidentiality and privacy—but not availability—with a small trusted computing base (TCB). In contrast, in this paper, availability of the device is one of our central goals.

The need to recover from software failure or compromise has existed for a long time, and solutions with properties

akin to dominance have emerged in other areas of computing. In the server and data center space, the Intelligent Platform Management Interface (IPMI) [22] allows data center administrators to install software updates on all servers irrespective of their state. Such solutions rely on separate control processors running separate operating systems (e.g., Intel Management Engine (IME) [23]) and accessing the network through either dedicated interfaces or special interfaces with sidebands [24]. Thus, while effective in the server domain, this approach appears unsuitable for resource-constrained IoT devices. More generally, the traditional last line of defense after a severe compromise has been for the administrator to assert dominance by taking physical control of the compromised device and cleaning it up or installing a new operating system on it. As the ratio of devices to administrators for IoT deployments is often orders of magnitude larger than in personal computing, even this approach becomes impractical.

This paper presents CIDER, a system that implements dominance for IoT deployments. CIDER allows the administrator to specify a firmware update and ensures that the update will be deployed and executed on all devices within a time-bound. This includes compromised devices executing adversarial code that is actively trying to avoid being evicted from the device. CIDER works with existing IoT hardware and requires only minimal changes in the hardware configuration and firmware.

A key design choice in CIDER was to avoid concurrent execution of trusted and untrusted code. CIDER device code runs during boot immediately after a device reset. After ensuring that the device firmware is in accordance with policy, CIDER enables simple hardware protections and relinquishes control to the firmware until the next reset. This design obviates the need for runtime isolation and the complex security hardware and software needed to support it (e.g., ring protections, memory management units, virtualization support, trusted execution environments). The complexity and limitations of runtime isolation have given rise to critical implementation bugs [25–27] and a multitude of side channels including speculative execution attacks [28–30]. While most processors contain hardware support for runtime isolation, CIDER does not use it, and it is not part of its TCB.

Instead, CIDER uses two much simpler hardware primitives to enforce isolation in time: *latches* and an *authenticated watchdog timer (AWDT)*. A latch is a protection that software can enable but not disable. Once enabled, latch protection remains on until the next reset. Latches allow CIDER to protect itself such that the firmware cannot remove the protections even though it is in full control of the device.

The AWDT is a new primitive that allows CIDER to regain control of a device from the firmware unconditionally by forcing a reset. The AWDT behaves like a conventional watchdog timer in the sense that it resets the system if the watchdog is not periodically serviced. However, in contrast to a conventional watchdog, an AWDT requires cryptographically protected keepalive messages issued by the remote administrator to defer the platform reset. Hence, it cannot be serviced independently by local firmware. We show how to implement an AWDT by repurposing simple existing hardware.

We have implemented our CIDER prototype on the SolidRun HummingBoard Edge (HBE) [31], the Raspberry Pi Compute Module 3 (CM3) [32], and the STMicroelectronics Nucleo-L476RG (NL476RG) [33], representing high-, middle-, and low-end IoT devices respectively, while addressing various implementation challenges in each of the device families. Our evaluation shows that CIDER does not interfere with existing operating systems or bare-metal apps that typically run on top of these platforms, and introduces minimal overhead in a reasonable production setting.

The goal of ensuring availability makes the software TCB of CIDER larger than that of other trusted computing systems that limit themselves to confidentiality and integrity [16–21]. The CIDER device code comprises around 23k lines of code (LoC). This includes device initialization, a storage driver and a small networking stack. We isolate the networking code to prevent network-based attacks from compromising CIDER. We shield the remaining core CIDER code from the adversary through isolation in time and by checking the integrity of all inputs using the formally verified High-Assurance Cryptographic Library (HACL) [34].

In summary, this paper makes the following contributions:

- We propose *dominance*, a new trusted computing primitive that allows a remote administrator to unconditionally recover and configure a device even after a complete compromise of the device firmware.
- We design CIDER, a practical system that brings dominance to IoT deployments without disrupting normal device operations. Our design avoids the need for runtime isolation which allows for a significant reduction in the hardware complexity and software TCB.
- We introduce the AWDT as a new hardware construct and present practical AWDT implementations in existing hardware and in software.
- We demonstrate how large amounts of off-the-shelf driver code can be used safely by shielding them from adversarial inputs.
- We have prototyped CIDER on three popular IoT devices. Our evaluation shows that CIDER is compatible with existing firmware and introduces a tolerable boot-up delay and a negligible runtime overhead.

II. BACKGROUND

A typical IoT system consists of multiple components. A possibly large number of *devices* that may be geographically dispersed interact with the physical world through *sensors and actuators*. The devices connect over the internet to one or more back-end servers (the *hub*) which are often located in the cloud, but could also be managed by an enterprise. The hub may store and analyze the sensor data sent by the devices and send instructions to the devices. In addition to these application-specific interactions, the connection between the hub and the devices may be used for device maintenance such as firmware updates. The hub exposes interfaces that allow the owners or

users of the IoT system to access the data gathered by the devices or to configure the system.

IoT devices are simple computers, typically equipped with various peripheral sensors (e.g., thermometers, cameras or accelerometers) or actuators (e.g., traffic light controls or motors). Low-end IoT devices may be built around a low-power system-on-chip (SoC) consisting of a single-core microcontroller running at moderate clock rates (e.g., tens of MHz) and small amounts of RAM (e.g., tens to hundreds of kB) and flash (e.g., hundreds to thousands of kB). Such devices typically run an application either bare-metal or on top of an embedded operating system such as mbed OS or FreeRTOS. High-end devices may feature multicore 64-bit CPUs running at GHz clock rates, several GBs of DRAM, external storage devices and a full operating system such as Linux, allowing them to run multiple applications.

A. IoT Examples

This section presents three IoT scenarios. Our goal is to provide examples of systems to which CIDER could be applied that are both concrete and representative of broader classes of IoT applications.

Air-quality monitoring system. Several organizations have deployed air-quality sensors in many locations across the globe. The sensors (i.e., IoT devices) measure various types of air pollution (e.g., ozone, PM2.5, carbon monoxide) and send the measured data to a server (i.e., hub). The hub may process the raw data in several ways (e.g., analytics, visualizations). The results can be accessed through web interfaces [35].

Traffic monitoring and control system. Traffic cameras and other traffic sensors have been deployed in many cities. The cameras may (or may not) process the images to infer traffic density, traffic jams or accidents. The raw or processed sensor data are sent to the hub which gathers and stores the sensor data and can run various analytics jobs. This can enable applications in which the hub computes traffic light settings that optimize global traffic flows and sends these settings to internet-connected traffic lights [36].

Remote elevator inspection. The devices are microcontrollers in elevators. In addition to running the elevators, the devices periodically gather readings from various sensors and send them to the hub. The hub can run various analytics jobs to identify elevators that may need a more thorough inspection by maintenance personnel [37].

Discussion of examples. At one extreme, pure sensor devices simply read the current values from peripheral sensors (e.g., air quality sensors) and send them directly to the hub. In the absence of an internet connection, these devices are useless. At the other extreme are hybrid devices such as the elevator controller that have IoT functions (e.g., sending diagnostics information to the hub) in addition to operational functions (e.g., running the elevator).

B. Building Blocks

This section provides background on components that we use to build CIDER.

Latches. A hardware latch is a simple state machine with only two states: {`open`, `locked`}. Its initial state is `open` and software can cause it to transit into the `locked` state (e.g., by writing to a hardware register). However, only a device reset will cause the latch to transition back into the `open` state. Each latch has an associated security function which is enabled if and only if the latch is in the `locked` state. We are interested in two types of latches that operate on persistent storage:

- A read-write latch (RWLatch) that once applied, blocks any *read* or *write* access to one or more storage regions.
- A write latch (WRLatch) that once applied, blocks any *write* access to one or more storage regions (read access is allowed).

CIDER uses RWLatches to protect per-device secrets and WRLatches to protect its code against unauthorized modification or deletion.

Orderly reset. CIDER requires that device reset and power-on provide a clean-state environment in which early boot code can execute deterministically, regardless of the actions of software that was running prior to the reset. We assume this behavior for CPUs. However, if a CPU is embedded in a platform with additional active devices (e.g., devices that can bus-master or reset the main CPU), then these devices must also be reset when the main CPU is reset. The resets of latched devices must be tightly coupled to resets of the main CPU.

Attestation. Attestation in CIDER is based on the Device Identifier Composition Engine (DICE) [38]. DICE supports device identity and attestation requiring only minimal hardware support. To support DICE, a device must be equipped with a 256-bit *unique per-device* secret—K_{PLATFORM}—which must be read-latchable. Trusted early-boot code uses this secret to enable untrusted code that may run subsequently to perform attestations. The hardware requirements of DICE are far simpler than those of alternatives such as TPM-based attestation.

In a nutshell, DICE code running during early boot reads K_{PLATFORM} and then latches it so that it becomes inaccessible to later software. DICE then uses a deterministic key-generation algorithm to create two asymmetric key pairs: the *DeviceID* key pair and the *Alias* key pair.

The DeviceID key pair is derived solely from K_{PLATFORM} and remains the same for the life of the device. The Alias key pair is derived from K_{PLATFORM} *and* the hash of the device firmware. Thus, the Alias key pair will change if the firmware is updated. DICE uses the DeviceID private key to certify the Alias public key and the hash of the device firmware. Before passing control to the firmware, DICE deletes K_{PLATFORM} and the DeviceID private key from RAM and registers, but passes the Alias private key and the Alias public key certificate on to the firmware. The firmware can use these keys to make attestation claims to a server by signing a server-generated nonce.

Entropy source. We require a source of entropy such as a true random number generator (TRNG) in order to generate nonces that an adversary cannot predict.

III. Adversary Model

We model a realistic and powerful *remote* attacker who tries to hack into the devices taking advantage of firmware vulnerabilities or configuration errors such as weak passwords. Attacks requiring physical access or proximity lie outside the scope of this paper, as we focus on scalable attacks.

The device firmware, while originally benign, is subject to exploitation by the adversary, resulting in the device executing the adversary's code. The adversary's capabilities are only restricted by the properties of the hardware which we assume to work correctly (i.e., according to specification). For example, the adversary is neither able to overwrite early boot code in Read-Only Memory (ROM) nor revert hardware latches without device resets. In addition, we assume that at least a critical subset of the CIDER code works correctly and is free of vulnerabilities. With a focus on device security, we assume the hub is trusted and secure. Securing the hub is out of this paper's scope.

The adversary may also attempt to eavesdrop on, tamper with or block the communications between devices and the hub with the goal of obtaining sensitive information, controlling the devices indirectly and preventing the hub from controlling or recovering devices, respectively. However, we assume that the adversary cannot block the communication indefinitely because long-lasting attacks can typically be detected and remediated by network operators. Symantec observed during Q3 of 2015 that less than one percent of network layer DDoS attacks lasted more than 24 hours [39].

A. Problem Statement

Our goal is to enable the hub to *unconditionally* recover control of all managed devices even after a complete compromise of the device firmware: a property commonly called *availability*. With control recovered, the hub may subsequently issue firmware updates to patch the vulnerability or change the security settings that led to the exploit and evict the adversary from the device. It may further request evidence from the device that the updates have been applied correctly.

Control recovery is challenging because an attacker may execute his/her code at the highest privilege and refuse to cooperate with the hub. However, once this challenge is solved, the security benefit of control recovery can be amplified by what the hub can do after regaining control.

Recovery from a complete software compromise inevitably requires hardware features or even new hardware constructs. For the sake of practicality, the hardware features we rely on must be either readily available on popular IoT devices or easy to obtain and integrate. More precisely, our design must be implementable on existing, unmodified mass market chips (i.e., microcontrollers, storage), as it is very hard to achieve broad adoption of custom chip designs. In contrast, modifications or extensions at the board level are much easier to implement and mass deploy. Many IoT boards have various hardware extension interfaces precisely to allow the user to attach additional hardware to the board. We take advantage of this option in a variety of ways.

In addition, any security mechanism we introduce should not significantly interfere with the functionality and performance of the existing firmware. We also should neither change the deployment model (e.g., increasing manual effort) nor require hardware that would significantly increase the cost of the devices.

IV. Design

In this section, we define *dominance* and identify two simpler primitives, *gated boot* and a *reset trigger*, that are sufficient to implement it. We present a simple secure design that achieves dominance but has several usability problems. The next section will describe the complete CIDER design which resolves these problems.

A. Defining Dominance

Definition. The hub *dominates* a device if the hub can choose *arbitrary code* and force the device to run it within a *bounded amount of time*.

In the example of a smart traffic control system [36], the *bounded amount of time* might be 1 hour, and the *arbitrary code* can either be the patched firmware with vulnerabilities fixed or a temporary operation routine (e.g., flashing red lights).

We decompose dominance into two simpler but related components, gated boot and reset trigger, each designed to address one functional requirement.

Gated boot. Gated boot ensures that the device will boot firmware that is authorized by the hub *at that time*. If, no such firmware is on the device at boot time (e.g., because the hub demands a newer version of the firmware or because the firmware was compromised), gated boot will first obtain and install an acceptable firmware version on the device before booting into it.

Reset trigger. After gated boot, the firmware has complete control over the device (subject only to hardware constraints). As the firmware was chosen by the hub, it can, in general, cooperate in performing regular maintenance tasks such as installing firmware updates requested by the hub. However, if the device is taken over by an attacker, this will not be the case. In order to prevent the attacker's code from running on the device indefinitely, the hub needs a mechanism to force a device reset (i.e., a *reset trigger*). This will preempt the firmware and invoke gated boot, which can examine and update the firmware as requested by the hub.

B. Security Primitives and Protocols

CIDER uses cryptography to provide communications security and endpoint authentication and attestation. The essential security building-blocks and protocols are described in this section.

Hub authentication. A hub public key is installed in the device during initial provisioning, and devices perform signature verification to ensure that messages from the hub are authentic.

Device authentication and attestation. CIDER devices are furnished with globally unique DICE secrets (i.e.,

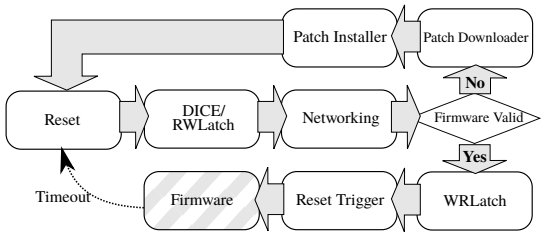

Fig. 1: An overview of a simple dominance design. The firmware cannot compromise the confidentiality and integrity of the system. However, this design has usability problems (frequent resets) and large attack surface (exposed networking stack). The improved, complete CIDER design is presented in Figure 2.

K_{PLATFORM}). The hub stores the derived DeviceID public key for each device during provisioning. This allows CIDER devices to authenticate themselves using their Alias key pairs and Alias key certificates.

Replay protection. CIDER protects all relevant hub-to-device and device-to-hub messages from replay attacks using nonces.

C. A Simple Dominance Scheme

Figure 1 displays a dominance scheme designed to be as simple as possible. However, it is a secure design under our threat model (§III) and protects itself from the firmware with the necessary hardware features.

Gated boot. Gated boot is implemented in the software that runs immediately after a reset. Some devices are hardwired to run small amounts of code in ROM first. In those cases, the gated boot code runs immediately after the ROM code.

The first task of gated boot is to run DICE to derive the DeviceID and Alias key pairs. This includes applying a RWLatch such that any attempts to read or write K_{PLATFORM} will be blocked until the next device reset. The next step is to ascertain if the hub authorizes the firmware that is currently on the device to run. Gated boot computes a cryptographic digest of the firmware's initial binary (`digest`), requests a `nonce` from the hub and sends `<digest|nonce>` signed by the DeviceID private key to the hub. While this only ensures the integrity of the initial binary, firmware features such as dm-verity [40] can extend integrity assurances to large parts of the firmware.

If the hub approves the received firmware `digest`, it replies with `<"OK"|nonce>` signed by the hub private key. Upon receiving the OK message, gated boot secures the device before transferring control to the firmware. In particular, it sets the WRLatch for the parts of storage where CIDER's code and data reside and enables the reset trigger which guarantees that CIDER will regain control within a bounded time interval. With these hardware protections enabled, gated boot loads the firmware's initial binary and transfers control to it.

If the hub does not approve the firmware, it replies with `<patch-id|nonce>` (signed by its private key), and gated boot will first download the firmware update from the hub according to the `patch-id` and use it to replace the old firmware on storage. In both cases, the net effect of gated boot is that the device will boot a firmware image that is approved by the hub. This behavior differs from prior work. Secure boot [41] will not boot at all if the firmware is unexpected. Authenticated boot [42] will boot any firmware and simply report to the hub what that firmware was.

Reset trigger. The simplest form of a reset trigger is a very simple timer that is akin to many existing watchdog timers. Once the timer is set, it cannot be disabled or deferred. When it expires, it resets the platform (unless the platform resets itself before that). The reset will invoke gated boot again.

D. Limitations

This simple design has several limitations which we will resolve in the next section:

Disruptive resets. If the reset trigger causes frequent and uncoordinated device resets, it may interrupt the device during a critical operation or cause in-memory state to be lost. Many IoT applications may not be able to tolerate this.

Boot delay. Gated boot adds a network interaction with the hub to every boot. This adds a noticeable delay.

Networking stack. The gated boot code includes a networking stack which, being large and exposed to attacks from the network, is a potential threat to the integrity of the CIDER TCB.

The next section will present an improved design that avoids these shortcomings. We will address the third problem by isolating the networking stack. We will solve the first two problems by making disruptive resets and network interactions at boot time rare events that should only occur under exceptional circumstances such as when the device is indeed compromised or there is a firmware update. The key to achieving this will be to enlist the help of the untrusted firmware. This help includes obtaining cryptographic tokens from the hub that allow forced resets and network interactions at boot time to be avoided. Failure to cooperate will result in a reset, invocation of gated boot and, possibly, the installation of a firmware patch.

V. AN IMPROVED DESIGN

This section presents the complete CIDER design. We describe how CIDER solves each of the three problems of the simple design and finally summarize the overall workflow of CIDER.

A. Avoiding Network Interactions at Boot Time

The basic version of gated boot has to contact the hub over the network each time the device boots, which can easily add seconds to an otherwise very fast boot sequence. Fortunately, the network interaction can be avoided completely under normal circumstances by offloading it to the firmware, which can overlap hub-communication with other activities, or perform it at a time when it is not disruptive.

Cooperating firmware may proactively fetch an authorization from the hub, named a `BootTicket`, that allows CIDER to boot the firmware directly without contacting the hub during the next boot. To enable this, gated boot generates a nonce (the `boot-nonce`) and WRLatches it, such that the firmware can read but not modify it. After control has been transferred

to the firmware, the latter includes the boot-nonce in a DICE-attested request to the hub for a BootTicket. In other words, the firmware can request a BootTicket by sending <digest|boot-nonce> signed by the Alias private key as well as the Alias public key certificate signed by the DeviceID private key.

If the firmware digest is within hub policy, the hub signs a BootTicket which includes the boot-nonce and sends it to the device, where the firmware will save it to unprotected persistent storage. After the next reset, gated boot will find the BootTicket, verify its signature and compare the boot-nonce from the BootTicket to the boot-nonce that it had originally generated and WRLatched the last time it ran. It will also compare the current firmware digest with the digest from the previous boot. If all tests succeed, it will start the firmware immediately, omitting the attestation step and the associated network interaction. Otherwise, or if no BootTicket exists, gated boot will fall back to the behavior described in the previous section including a network-based attestation step. Gated boot replaces the boot-nonce with a new random number on every boot. Thus, old boot-nonces simply become invalid and cannot be replayed.

If the firmware digest does not comply with hub policy, a new firmware version must be installed. The hub may choose to do this in collaboration with the current firmware. Alternatively, it may simply refuse to issue a BootTicket which will cause gated boot to install a new firmware version at the next reboot using its own update mechanism.

With this optimization, gated boot will only incur the cost of network transactions if the previous execution of the firmware failed to obtain a valid BootTicket. The latter should only happen only if 1) the firmware is uncooperative (either because of an attack or software failure), 2) the hub refuses to let the current firmware version continue its execution on the device or 3) a prolonged interruption in network connectivity. The security trade-off of the optimization is that, after obtaining the BootTicket, the firmware will be able to survive one more reset, even if the hub discovers that it is outdated or compromised. However, this can be compensated by shortening the timeout period of the reset trigger.

B. Avoiding Uncoordinated Resets at Runtime

As noted in §IV-D, the simple reset trigger can seriously disrupt the device if it frequently causes unexpected resets. Here, we describe a reset trigger whose disruption is not worse than that of a regular software update (e.g., patch Tuesday). For cooperating devices, resets can be pre-announced and should only happen when the hub requires a firmware update but does not trust the existing firmware to apply it. We assume this to be a rare event.

Authenticated watchdog timer (AWDT). We define an AWDT as a new abstract hardware device and describe how CIDER uses an AWDT to implement a reliable, non-disruptive reset trigger. Since the AWDT is a new device, there is no off-the-shelf implementation of it. We will present our own AWDT implementations that repurpose existing hardware in §VI.

An AWDT is a device that can be programmed to trigger a reset after a given amount of time unless the reset is deferred in an *authenticated* way. Formally, let AWDT_Init(T, K) be the function that starts the AWDT. The AWDT will trigger a reset T seconds after AWDT_Init is called. The K parameter is a public key for signature verification. AWDT_Init is a latched operation. Once AWDT_Init has been called, its effect cannot be undone nor can it be called again until the next reset.

However, AWDT expiration can be deferred in an authenticated way. If the AWDT is provided with a voucher signed with the private key that corresponds to the public key K, it will extend the time-to-reset by the number of seconds stated in the voucher. We call such a voucher a DeferralTicket. It has three components: a nonce (to avoid replays), the number of seconds by which the reset should be deferred and a signature over the other components. The second AWDT interface AWDT_GetNonce returns the nonce while the third interface AWDT_PutTicket hands over a DeferralTicket to the AWDT. AWDT_PutTicket checks the nonce and the signature on the DeferralTicket and, if both tests succeed, extends the time-to-reset.

AWDT in CIDER. CIDER gated boot calls AWDT_Init(T, K) to initialize the reset trigger, where K is the hub public key and T is an appropriate timeout period (e.g., one day). Once control has passed to the firmware, it is the task of the firmware to obtain DeferralTickets to prevent the AWDT from resetting the device. The mechanics of this are analogous to those of obtaining a BootTicket. The firmware calls AWDT_GetNonce and sends a DICE-attested request for a DeferralTicket to the hub. The hub decides whether to issue a DeferralTicket based on the firmware hash and the DeviceID. If the hub is satisfied, it issues a DeferralTicket and sends it to the device. The firmware can then forward the ticket to the AWDT by calling AWDT_PutTicket.

If the hub is dissatisfied with the attestation, it may try to coordinate an orderly update and restart with the existing firmware. This could involve issuing it a shorter-duration DeferralTicket to enable the device to run until the next convenient time to apply the update and restart the device.

The main difference between an AWDT and a conventional watchdog timer is the servicing mechanism. Regular watchdog timers are serviced through unauthenticated software actions such as writing a constant to a register. In our setting, the untrusted firmware can delay the reset from regular watchdog timers indefinitely by servicing them. In contrast, an AWDT can only be serviced with the help of a fresh DeferralTicket issued by the hub.

The gated boot code informs the firmware about the initial AWDT time-to-reset. The firmware can also see the deferral time in every DeferralTicket it obtains from the hub for the AWDT. Thus, the firmware knows when the AWDT will reset the device and can make appropriate preparations, such as hibernating itself, putting the environment into a safe state and/or restarting the device at a time that is least disruptive.

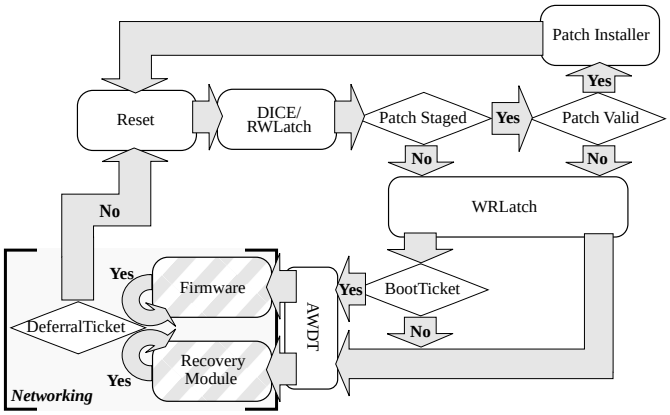

Fig. 2: An overview of the complete CIDER design. Neither the firmware nor the recovery module (with its isolated networking stack) can compromise the confidentiality and integrity of CIDER.

C. Isolating the Networking Stack

We isolate the networking stack together with a small amount of control logic into a separate *recovery module*. The exact security implications of this step are discussed in §VII. CIDER gated boot will invoke the recovery module if no valid `BootTicket` is present. The task of the recovery module is to obtain a `BootTicket` or a firmware update from the hub.

The recovery module is a separate binary. We isolate it using the same isolation-in-time mechanism that CIDER uses to protect itself from the firmware. Before invoking the recovery module, CIDER activates all latches and the AWDT, loads the binary, places its input parameters into memory and transfers control to it. The input parameters include the digest of the firmware as well as a set of DICE credentials for the recovery module including an Alias key pair.

When invoked, the recovery module performs a DICE attestation of the firmware digest to the hub and requests a `BootTicket`. If the firmware digest is in accordance with hub policy, the hub returns the `BootTicket`. Otherwise, the hub sends a firmware update. Either way, the recovery module saves the result to persistent storage an returns control to CIDER gated boot by resetting the device. All network communications are secured with nonces and signatures.

D. An Improved Dominance Scheme

Figure 2 summarizes the complete CIDER design. It integrates the enhancements described in this section into the basic design of Figure 1.

After a reset, CIDER gated boot initializes DICE. This includes reading K_{PLATFORM}, computing the DeviceID and Alias keys and RWLatching K_{PLATFORM}. If a firmware update was staged during the previous run, CIDER will validate and install it and reset the device. This mechanism can also update CIDER itself by replacing its binaries or configuration data on the storage device. This also includes reprovisioning CIDER with a new hub public key (e.g., to change device ownership) or with new network credentials.

In the absence of a staged firmware update, CIDER will WRLatch its code and data and look for a `BootTicket`. If a valid `BootTicket` is present, CIDER will enable the AWDT and transfer control to the firmware. In all other cases, CIDER will apply the same protections and invoke the recovery module. During its execution, the firmware (or possibly the recovery module) will periodically interact with the AWDT and the hub to serve the AWDT valid `DeferralTickets` to postpone the reset.

E. Safe Mode

Both gated boot and the AWDT assume the ability to communicate with the hub for `BootTicket` fetching and AWDT servicing. However, our threat model allows for the network to become temporarily unavailable, e.g., because of a DDoS attack. There are a range of options for CIDER to handle this case, representing various trade-offs between availability and integrity. The optimal choice depends on the application. For example, for pure sensor devices that merely send data to the hub while having no offline function, it may be acceptable to just wait until gated boot can reach the hub again.

For hybrid devices with offline functionality, booting into a "safe mode" firmware image that is stored on the device and protected by a WRLatch may be the best option. The safe mode firmware allows the device to perform its offline tasks but does not enable network or other functionality that could cause the device to be compromised (e.g., by keeping the network interface controller (NIC) off). For example, safe mode elevator firmware may allow the elevator to operate, but not support remote diagnostics. This does not appear to give up any functionality since, by assumption, the connection to the hub is not available. After some time, the AWDT or the safe mode firmware itself may reset the device to give gated boot another opportunity to contact the hub.

VI. IMPLEMENTATION

We have implemented CIDER on three popular IoT platforms:

- SolidRun HummingBoard Edge (HBE), priced ~$240, representing a high-end, powerful computing board;
- Raspberry Pi Compute Module 3 (CM3), priced ~$35, in conjunction with the CM3 IO Board ($120), representing a mid-end, generic multi-purpose board;
- STMicroelectronics Nucleo-L476RG (NL476RG), priced ~$15, representing a low-end, resource-constrained board.

We separated the implementation of CIDER into a platform-independent part and a hardware abstraction layer (HAL). The HAL for each platform implements a common interface on the hardware available on the platform. The platform-independent part implements the logic flow as described in §V-D over the HAL interface and consists of about 6,300 lines of C code. Most of this code is made up by the crypto library (3,600 LoC). We used the digital signature algorithm (Ed25519) and SHA2-256 components of the formally verified High-Assurance Cryptographic Library (HACL) library [34, 43]. We begin by describing our hub and AWDT implementations which apply to

all three platforms. After that, we describe the device-specific aspects of the implementation for each of our target platforms.

A. Hub

We built a simple hub prototype to test CIDER end-to-end. The hub prototype consists of 2,500 lines of C# code based on Azure Functions [44], using the Bouncy Castle crypto library [45] and Azure SQL Database for state storage [46].

B. AWDT

The AWDT is a new primitive. No existing MCU implements it. However, we believe that adding such functionality would be easy. For example, the HBE's MCU already has all major building blocks (i.e., TRNG, crypto accelerator, conventional watchdog timer), and would only require simple control logic to implement an AWDT.

Not being able to change the MCU, we present two alternative AWDT implementations: 1) using simple hardware that we attach to the main board and 2) using only software.

Hardware AWDT. We implemented an external authenticated watchdog timer (eAWDT) using a separate STMicroelectronics Nucleo-L053R8 (NL053R8) board [47], featuring a 32 MHz STM32L053R8 MCU (ARM Cortex-M0+). The STM32L053R8 cost less than $3 at volume. We chose the NL053R8 board to simplify our implementation work. The hardware cost could be reduced to less than $1 by combining a cryptographic co-processor such as the Microchip ATECC608A [48] (for nonce generation and signature verification) with an 8-bit microcontroller such as the Microchip ATtiny412 [49] (for control logic and universal asynchronous receiver/transmitter (UART) communication).

We used 200 lines of C code to implement the three functions of the AWDT interface (i.e., `AWDT_Init`, `AWDT_GetNonce` and `AWDT_PutTicket`). In addition, we had to include our crypto library (3,600 lines of code) for the signature check on the `DeferralTicket`. The main board calls these functions by communicating over the UART interface.

`AWDT_Init` sets the initial counter value and initiates a periodic timer interrupt (triggered every 1 s) that executes a callback function to decrease and check the counter value. When the value becomes zero, it signals one of the general-purpose input/output (GPIO) pins of the NL053R8 board. This pin should be connected to the reset pin of the main board to reset it from the outside. The eAWDT then resets itself as well by executing `SYSRESETREQ`. `AWDT_GetNonce` uses the NL053R8's TRNG to generate cryptographically secure nonce values.

The simplicity of the AWDT interface provides strong protection of the AWDT against attacks from the main board and vice versa.

Software AWDT. As an alternative to the eAWDT, we also implemented the AWDT in software running in TrustZone on the HBE and the CM3 and using the Memory Protection Unit (MPU) on the NL476RG. This software implementation relies on runtime isolation. However, as the AWDT contains no secrets and, thus, does not require confidentiality, it is not subject to known side channel or speculative execution attacks.

Our software implementation uses minor variants of the 200 lines of C code running on the eAWDT. On processors that support TrustZone, we implemented the AWDT code as a pseudo trusted application (PTA) on OP-TEE [50] in TrustZone. Our code programs an existing, regular timer whose control registers can only be accessed from within TrustZone. On the NL476RG which lacks TrustZone support, we used the MPU to isolate our AWDT implementation. Due to space limitations, the rest of this paper discusses only the eAWDT.

C. SolidRun HummingBoard Edge (HBE)

Our HBE features an NXP i.MX6Quad (ARM Cortex-A9) quad-core processor running at 1 GHz. This processor supports a variety of security features, including protected key storage, a cryptographic accelerator, ARM TrustZone, a TRNG, and a secure real-time clock. The board also has a gigabit Ethernet controller, 2 GB of DRAM and an 8 GB Embedded Multi-Media Card (eMMC).

We integrated our boot code into the U-BOOT [51] Secondary Program Loader (SPL). U-BOOT is a popular boot loader. The SPL is a small subset of U-BOOT. Its purpose is to initialize the hardware to the point that full U-BOOT can run. The SPL for the HBE has to turn on DRAM and processor caches, load the full U-BOOT binary from eMMC into DRAM and transfer control to it. We inserted the CIDER boot code into the SPL after the end of the hardware initialization but before the loading of the U-BOOT binary. This freed us from having to write device-specific initialization code and provided us with a basic runtime environment and an MMC storage driver. Table I displays the line counts. As CIDER's protections (i.e., latches, AWDT) are turned on in the SPL, CIDER is protected from U-BOOT and any binaries it might load (e.g., Linux).

Networking. For convenience, we built the recovery module out of a stripped-down version of U-BOOT. U-BOOT contains a driver for the i.MX6 Ethernet controller and a simple networking stack, which we augmented to support TCP with a patch [52]. We removed all unnecessary U-BOOT components.

RWLatch protected key storage. The Cryptographic Acceleration and Assurance Module (CAAM) of the HBE's MCU (i.MX6) provides confidentiality and integrity protection for critical data blobs with a key that is only accessible by the CAAM itself. Critically, use of the key is gated by a RWLatch. The `PRIBLOB` bits in the Security Configuration Register (`SCGFR`) must be 0 for the key to be readable. Software can set the bits to 1 at any time. Once set to 1, the `PRIBLOB` bits will retain this value until the next reset. We use this mechanism to protect the device private key. At each boot, the CAAM will decrypt the private key for CIDER. Gated boot then sets the `PRIBLOB` bits to 1, disabling access until the next reset.

WRLatch protected code and data storage. The eMMC standard supports *power-on write protection*. This feature allows software to instruct the eMMC device to make parts of storage read-only until the next reset. All storage of the eMMC device can be write-protected in this way at 8 MB granularity.

Unfortunately, the HBE does not couple eMMC reset tightly to processor reset. Instead, it allows the eMMC reset pin to be controlled by software via GPIO. We solve this problem with the help of the SECURE_WP_MODE feature of the eMMC 5.1 standard that allows power-on write protection to be locked such that resets do not remove it [53]. The lock and unlock commands are authenticated with a secret number. At installation time, CIDER sets this number and keeps an encrypted copy (using the RWLatch). When setting power-on write protection, CIDER also issues the lock command to the eMMC device to guard against spurious resets. When needed, CIDER removes the power-on write protections by issuing the unlock command and subsequently resetting the eMMC device by signaling its reset pin via GPIO.

Authenticated watchdog timer. We connected our eAWDT board to the mikroBus interface on the HBE. We connected the eAWDT's reset wire to the reset pin (RST_n) of the mikroBus. Furthermore, the HBE exposes the HBE's UART2 through the mikroBus which we connect to the eAWDT's UART interface.

Firmware support. We added CIDER support to Windows 10 IoT Core [54] and Debian [55] firmware by including an application in the firmware that communicates with the AWDT and obtains `BootTickets` and `DeferralTickets`. The application is periodically woken up at the desired `DeferralTicket` fetch interval. It calls `AWDT_GetNonce` on the eAWDT, requests a `DeferralTicket` from the hub and calls `AWDT_PutTicket`. The application runs as part of the untrusted firmware and is not part of the CIDER TCB. The Windows IoT version is a Universal Windows Platform (UWP) application consisting of 750 lines of C# code. The Debian version is a Posix application comprising about 1,100 lines of C code. In both cases, the majority of the code supports the network communication with the hub.

D. Raspberry Pi Compute Module 3 (CM3)

The CM3 board features the Broadcom BCM2837 architecture, including an ARM Cortex-A53 quad-core CPU running at 1.2 GHz and 1 GB DRAM. It also has a 4 GB eMMC storage device and a TRNG, but lacks a RWLatch. As in the case of the HBE, we integrated the CIDER boot code into the U-BOOT SPL.

Networking. The CM3 has no built-in networking hardware, so we connected a USB Ethernet Adapter (around $1.5) to it. We reused the networking code from the HBE implementation—except for a different NIC driver.

RWLatch protected key storage. The CM3 lacks a RWLatch. To solve this problem, we bought an OPTIGA SLB 9670 chip which supports the TPM 2.0 specification [12] at an extra cost of $2.09 and connected it to the CM3 IO board through GPIO pins. We used the internal non-volatile storage of the TPM to host the device private key. We used the commands `TPM2_NV_ReadLock` and `TPM2_NV_WriteLock` to implement the RWLatch over the internal storage of the TPM. We could also have used any other TPM 2.0 chip. For example, the AT97SC* series from Microchip can be as cheap as $0.85 per chip.

WRLatch protected code and data storage. We use the power-on write protection feature of the eMMC device. Unfortunately, the CM3 does not connect the eMMC's reset pin RST_n to any reset signal. Instead, RST_n is permanently connected to a pull-up resistor [56]. The result is that power-on write protection stays on even after a reset. We solved this problem by soldering a wire to RST_n and connecting it to the CM3 IO Board's RUN pin to let them share the same external reset signal.

Authenticated watchdog timer. The CM3 IO Board exposes 54 GPIO pins. We programmed GPIO 40 and 41 to carry the UART's transmit (Tx) and receive (Rx) signals and connected the corresponding wires of the eAWDT. We also connected the eAWDT's reset wire to the IO Board's RUN pin, allowing the eAWDT to reset both the processor and the eMMC device.

Firmware support. We added CIDER support to both Raspbian and Buildroot-based firmware [57, 58] by including the same ticket fetching application we had used for Debian on the HBE.

E. STMicroelectronics Nucleo-L476RG (NL476RG)

The NL476RG board features an STM32L476RG MCU (based on ARM Cortex-M4) running at 80 MHz with 1,024 kB flash memory and 128 kB SRAM, as well as a TRNG. Since U-BOOT does not run on the NL476RG board, we wrote CIDER as a bare-metal application. We modified the NL476RG's linker script to physically separate the flash memory regions for CIDER and the firmware, and flashed them independently.

Networking. Since the NL476RG board has no networking hardware, we connected an ESP8266 ESP-12E Wi-Fi module (around $1) to the board via the UART interface. The module includes a stand-alone networking stack that supports TCP/IP.

RWLatch protected key, code and data storage. We used the STM32L476RG MCU's firewall feature [59] as a RWLatch. The firewall makes it possible to block all access to specific address ranges until the next reset.

Right before transferring control to the firmware, CIDER configures the firewall to block all access to the flash memory segment storing its secret keys, code and data. The mechanics of the firewall require us to copy the code that enables it to an unprotected region. Any CIDER data that need to be read by the firmware (e.g., the `boot-nonce`) are also copied to unprotected storage before the firewall is enabled. Any attempt to access the protected region results in a device reset.

Authenticated watchdog timer. The NL476RG provides 47 GPIO pins. We chose PA9 and PA10 which can be used for UART Tx and Rx, respectively, and connected them to the eAWDT. The NL476RG also has a pin (NRST) to receive an external reset signal; we connected this pin to the eAWDT's reset wire.

Firmware support. Lacking an operating system, we inserted the ticket fetching code directly into the target applications while registering a timer interrupt handler to periodically execute the code. The code mirrors largely that of the Debian application.

Software TCB	LoC	Exposed	Defense
Boot module			
CIDER gated boot	2,700	×	Isolation in time, Firewalling
CPU & board init	4,700	×	Isolation in time, Firewalling
Device drivers	4,500	×	Isolation in time, Firewalling
Crypto lib	3,600	✓	Formal verification
Recovery module			
CIDER control logic	200	✓	Firewalling, Compartmentalization
Networking stack	5,200	✓	Firewalling, Compartmentalization
Networking driver	1,600	✓	Firewalling, Compartmentalization
Crypto lib	3,600	✓	Formal verification
***e*AWDT**			
CIDER control logic	200	✓	Firewalling, Simplicity
Crypto lib	3,600	✓	Formal verification

TABLE I: Software TCB of CIDER and the measures CIDER takes to secure it. The Line of Code (LoC) count is based on CIDER's prototype implementation on HBE.

VII. Security Analysis

In this section, we analyze the software TCB of CIDER. We enumerate its components and describe the techniques we used to secure it. For concreteness, we focus the discussion on the HBE variant of the CIDER implementation.

A. Summary of the Software TCB

The CIDER implementation consists of three discrete modules: the boot module, the recovery module and the AWDT. Each module contains the same formally verified cryptographic library. Table I lists the main components of each module.

Boot module. In addition to the core part of CIDER gated boot, the boot module contains device initialization code and device drivers from the U-BOOT SPL. This includes code for initializing DRAM and the CPU caches as well as MMC storage, UART and GPIO drivers.

Recovery module. The recovery module encapsulates the U-BOOT networking stack augmented with TCP and the HBE NIC driver. It also contains the U-BOOT MMC storage driver and a small amount of CIDER control logic for managing the interaction with the hub, including DICE attestation.

AWDT. The *e*AWDT software consists of code implementing the AWDT control logic as well as low-level code for accessing the UART and the TRNG.

B. Defending the Software TCB

The need to ensure device availability adds a storage driver, a networking stack, and device-dependent low-level initialization code to our TCB. As shown in Table I, these components add a substantial amount of code to CIDER. Much of the existing work on trusted computing has excluded this code from its TCB by making availability a non-goal. In this setting, a modest amount of cryptography and protocol code (e.g., SSL, encrypted disks) are sufficient to protect the confidentiality and integrity of data sent to untrusted storage and network devices. No attempt is made to ensure that these devices are functioning.

Rather than attempting to make the entire software TCB bug free—a daunting task in a setting like ours—CIDER uses several defensive techniques to ensure its integrity and availability even in the presence of bugs. At the core of our defense lies the following assertion: *A bug in code can only become an exploitable vulnerability if the attacker can influence the environment (e.g., function parameters, stack content, heap content) in which the code executes.* Regular software testing does not guarantee the absence of vulnerabilities, but it ensures that code will behave correctly under normal conditions. Attackers exploit vulnerabilities creating abnormal conditions by carefully controlling the environment (e.g., heap spraying, calling functions with unexpected parameters, concurrency).

Isolation in time. Isolation in time is a critical tool in restricting an attacker's influence on the environment in which CIDER executes. Execution begins with a reset which creates a clean-slate environment for CIDER to run in. No untrusted code runs until CIDER completes execution. Before transferring control to the firmware, CIDER write-protects its code and state, preventing untrusted code from changing them. CIDER also hides its secrets by zeroing out the memory used in gated boot and applying the RWLatch. The only information that explicitly flows from the untrusted code running on the device to CIDER gated boot are firmware updates and `BootTickets`, which will be rejected by CIDER unless properly signed by the hub.

Firewalling. Firewalling aims at reducing the code exposed to attackers, and hence, reducing the attack surface.

All messages from the hub are signed. After reading a `BootTicket` or firmware patch from the storage device (raw block read), CIDER checks its signature and ignores the input if the check fails. This limits the code exposed to adversarial inputs to a block read and a signature check. The signature check is performed by the formally verified HACL library. In summary, the only complex code in the boot module that is exposed to untrusted inputs is formally verified. The remaining code, which includes thousands of lines of potentially vulnerable off-the-shelf driver code, runs under nearly deterministic conditions and is shielded from untrusted inputs. A similar argument applies to the `DeferralTicket` validator in the AWDT.

Our *e*AWDT implementation is isolated on separate hardware that only shares two UART wires and a reset wire with the main device. The AWDT interface is very small and simple (i.e., `AWDT_GetNonce`, `AWDT_PutTicket`), resulting in a small attack surface.

Finally, CIDER runs the networking stack in a very limited way. It has no open ports or incoming connections. All network activity originates on the device and consists of TCP connections with the hub. Although an adversary may try to obtrude on such a connection by injecting packets at various protocol layers, many such attacks can be recognized and eliminated early by filtering out packets that do not meet the constraints of connections between a CIDER device and the hub.

Compartmentalization. The networking stack is the only component of CIDER with a non-trivial attack surface. CIDER isolates it such that even a potential exploit against the

networking stack will not compromise the device.

More precisely, the boot module treats the recovery module which contains the networking stack exactly like the firmware. Before invoking the recovery module, the boot module activates all protections (i.e., RWLatch, WRLatch, AWDT). The recovery module returns to the boot module by writing the hub's response to unprotected storage and invoking a reset. Thus, a compromised recovery module will not give an attacker any capabilities that he or she could not get from compromising the firmware.

An exploitable vulnerability in the networking stack would allow an adversary to take control of the device temporarily until the next AWDT-triggered reset. For invocations of the recovery module, CIDER sets the AWDT period based on the expected time needed to complete the network transaction, which can be much shorter than the period for the firmware (e.g., seconds to minutes vs. hours to days).

After the reset, the boot module and the *unaltered* recovery module will run again. The adversary may be able to re-infect the still-vulnerable networking stack until the vulnerability is patched. A possible mitigation involves having the Internet Service Provider (ISP) set router or firewall rules to block attack packets and allow CIDER to install the patched networking code.

This scenario (i.e., a vulnerability in a critical CIDER component that is exploitable in spite of several defense layers) should be extremely rare. It will interrupt device availability temporarily (i.e., until the ISP has blocked the attacker). However, even this worst-case scenario has a clear recovery path that does not require each device to be restored manually.

VIII. EVALUATION

In this section, we evaluate the practicality and performance of CIDER. In particular, we answer the following questions:

- Does CIDER interfere with existing IoT software given its extra resource consumption and periodic ticket fetching?
- How much delay does CIDER introduce to the device boot-up process in various scenarios?
- What is the runtime overhead due to the eAWDT on various devices?

A. Software Compatibility

The variety of use cases for IoT devices and the diversity of IoT firmware make it inherently challenging to argue that a technology such as CIDER is compatible with all possible IoT firmware. However, we can gain confidence empirically that CIDER is compatible with commonly used firmware that typically runs on the tested IoT boards.

To do this, we installed both CIDER and standard firmware containing an operating system (for the HBE and CM3) or a bare-metal app (for the NL476RG). We ran the battery of tests summarized in Table II on each of the boards without observing any abnormal behavior or other interference from CIDER.

In particular, we tested each device with two different types of firmware, and for each firmware, we tested two scenarios: 1) an extremely frequent ticket fetching policy (i.e., every 15 seconds) to stress the system over a short time period; and 2) a more realistic ticket fetching policy (i.e., every 4 hours) for a long-running system (i.e., 1 day). We did not observe abnormal behavior during any of the experiments. All the operations worked as expected.

B. Performance

We measured the device boot-time delay and runtime overhead introduced by CIDER. In the following experiments, the devices have to interact with the hub. Instead of running our own server to host the hub, we deployed our hub prototype in the cloud on Azure Functions (a serverless compute service) [44], simulating the actual management model of cloud-managed IoT devices. We used the gigabit Ethernet NIC on the HBE and Ethernet over USB for the CM3 to connect these boards to the internet. We connected the NL476RG to the internet through Wi-Fi using the ESP8266 module.

Boot-time delay. CIDER runs immediately after a device reset and thus, may affect the boot time of the device depending on whether a `BootTicket` is available or whether a firmware update is required. We measured how long CIDER takes to perform a gated boot before jumping to the firmware. On the HBE and the CM3, we used the ARM processor's Cycle Count Register for accurate measurement. We divided the CPU cycle count by the clock rate to obtain the elapsed time. Since the NL476RG's Cortex-M4 does not have a Cycle Count Register, we used a second NL476RG board to measure the boot time. We connected one GPIO pin of the measurement board to the reset pin (NRST) of the target board. We connected a second GPIO pin on the measurement board to a GPIO pin on the target board. To measure the boot time, the measurement board resets the target board and waits for code that we inserted into the beginning of the application to signal the second GPIO pin. We measured the time between these two events.

Table III shows the results for all three platforms. The *w/o* CIDER baseline configuration is the unchanged original device configuration. The remaining three measurements correspond to different gated boot scenarios. First, in the *w/ `BootTicket`* case, the device already has a `BootTicket` and can boot the firmware without having to access the network. The overhead arises primarily from from the crypto operations performed during DICE initialization and is most pronounced on the relatively slow NL476RG. This should be the standard case for devices that run cooperating firmware.

In the next case (*w/o `BootTicket`*), the device has no valid `BootTicket` because the firmware did not fetch it before the last reset. Thus, the CIDER recovery module has to obtain a `BootTicket` from the hub and reset the device. This is followed by execution of the *w/ `BootTicket`* case. The additional network-based attestation and reset increase the boot time.

In the last case (i.e., *w/ patch*), the hub has demanded a firmware update and it did not issue a `BootTicket` before the last reset. As a result, CIDER has to fetch a firmware patch from the hub via the recovery module, go through a reset to apply the patch, and then follow the steps of the previous case.

Device	Firmware	Ticket fetch	Operations	Normal?
HBE	Windows	15 seconds	Boot; Login; Install the InternetRadio app; Launch the app; Wait for 2 minutes; Terminate; Logout; Shutdown	✓
	IoT Core [54]	4 hours	Boot; Login; Leave the OS running for 24 hours; Logout; Shutdown	✓
	Debian [55]	15 seconds	Boot; Login; apt-get install radiotray; Launch the app; Wait for 2 minutes; Terminate; Logout; Shutdown	✓
		4 hours	Boot; Login; Leave the OS running for 24 hours; Logout; Shutdown	✓
CM3	Raspbian [57]	15 seconds	Boot; Login; wget <link-to-a-large-file>; Wait for 2 minutes; Terminate; Logout; Shutdown	✓
		4 hours	Boot; Login; Leave the OS running for 24 hours; Logout; Shutdown	✓
	Buildroot [58]	15 seconds	Boot; Login; mkfile -n 2g TestFile; Wait for 2 minutes; Terminate; Logout; Shutdown	✓
		4 hours	Boot; Login; Leave the OS running for 24 hours; Logout; Shutdown	✓
NL476RG	FFT [60]	15 seconds	Reset; Leave the sound sampling and Fast Fourier Transform (FFT) analysis program running	✓
		4 hours	Reset; Leave the sound sampling and Fast Fourier Transform (FFT) analysis program running	✓
	TLC [61]	15 seconds	Reset; Leave the traffic light controller (TLC) running	✓
		4 hours	Reset; Leave the traffic light controller (TLC) running	✓

TABLE II: Tests performed to verify that the deployment of CIDER does not interfere with various firmwares and bare-metal applications.

Boot Config	HBE	CM3	NL476RG
w/o CIDER	0.98	1.25	0.01
w/ `BootTicket`	1.25 (+0.27)	1.73 (+0.48)	4.35 (+4.34)
w/o `BootTicket`	6.42 (+5.44)	8.61 (+7.35)	17.50 (+17.50)
w/ patch	15.60 (+14.60)	20.80 (+19.50)	30.20 (+30.20)

TABLE III: The boot-up time and (+delay) in seconds of the three IoT platforms with CIDER according to whether a `BootTicket` exists and whether a firmware patch has been staged.

Interval	HBE	CM3	NL476RG
1 min	0.28% (0.54%)	0.32% (0.97%)	0.64% (0.30%)
5 min	0.15% (0.33%)	0.09% (0.58%)	0.16% (0.26%)

TABLE IV: Runtime overhead of the SPEC CPU2006 and CoreMark benchmarks evaluated on the three IoT platforms with CIDER according to various `DeferralTicket` fetching intervals (geometric mean and standard deviation).

The extra delay is primarily caused by the time needed to download the firmware patch and install it. In the experiments, we downloaded a small patch of size 4 kB. Larger patches will require more time depending on the speed of the network connection and of the storage device.

Runtime overhead. The firmware has to periodically interact with the eAWDT and the hub to keep the eAWDT from resetting the device. This introduces runtime overhead. We measured the execution time of various benchmarking programs with and without CIDER enabled. On the HBE and CM3, we ran the SPEC CPU2006 benchmark suite [62] on Debian and on Raspbian, respectively. We cross compiled and ran all C and C++ SPEC applications, except for applications that the `arm-linux-gnueabihf` toolchain (version 7.3.0) failed to compile (`perlbench`, `dealII` and `soplex`) or that suffered from runtime errors due to incompatibilities (`gcc`, `zeusmp`, `omnetpp` and `sphinx3`). This left a total of 21 SPEC applications. Since both the HBE and CM3 have multi-core processors, we pinned the applications as well as the code responsible for the overhead, i.e., a daemon that fetches a `DeferralTicket` and interacts with the eAWDT, to the same core. This ensures that the overhead is not hidden through execution on an idle core. At the same time, this is likely to overestimate the overhead, as it is charged to a single core rather than being amortized across multiple cores.

Porting a diverse set of user mode applications like SPEC CPU2006 into the bare-metal environment of the NL476RG poses several difficulties. Instead, we used the CoreMark benchmark [63] developed to test the performance of MCUs. We added a periodic timer interrupt handler executing the code to interact with the eAWDT and the hub to CoreMark.

On all platforms, we ran each benchmark application without CIDER (baseline) and with CIDER using AWDT expiration periods of 1 minute and 5 minutes. The period of 1 minute is an extreme case to identify the worst-case performance overhead.

Overall, the overheads were negligible (Table IV). The numbers for the HBE and the CM3 are the geometric means (and standard deviations) over the running times of the SPEC applications. The numbers for the NL476RG are averaged over ten runs of the CoreMark application. Even for the unreasonably short AWDT period of 1 minute, the overheads were 0.28% (HBE), 0.32% (CM3) and 0.64% (NL476RG) on average. This was because the code interacting with the eAWDT and the hub is mostly idle and occasionally performs short UART and networking I/O tasks. For longer ticket fetching intervals, the overhead was near zero and disappeared in the measurement noise.

We examined the overhead sources by measuring the wall-clock times for a call to `AWDT_GetNonce` and `AWDT_PutTicket`, as well as the time required to fetch a `DeferralTicket` from the hub. We found no significant difference between the three IoT devices as the overhead is largely determined by external factors: UART and eAWDT speed, and network delay. We measured the following numbers on the CM3. On average, `AWDT_GetNonce` takes 3.431 ms (standard deviation: 0.002) and `AWDT_PutTicket` takes 0.014 ms (standard deviation: 0.002). The numbers are averaged over 1,000 runs. `AWDT_PutTicket` is fast because the eAWDT will verify the ticket asynchronously; the caller does not have to wait for the eAWDT's response. The time to fetch a `DeferralTicket`, which heavily depends on network conditions and server load, was 635.29 ms on average (standard deviation: 565.51).

Although all three devices display similar wall-clock overheads, the HBE and CM3 can do useful work while the CIDER

thread is blocked waiting for a response from the network or the eAWDT. The operating system can simply schedule a different thread. In contrast, our simple, bare-metal NL476RG implementation spends this time busy-waiting. However, ticket fetching is a rare operation and does not cause significant overhead even in this case.

IX. Related Work

Several industry standards—the Intelligent Platform Management Interface [22], the Data Center Manageability Interface [64], the Redfish Scalable Platforms Management API Specification [65] and Intel Active Management Technology [66]—enable the efficient remote configuration and monitoring of servers. This functionality is akin to dominance. However, these systems are not appropriate for the IoT space, as they require a separate co-processor (with a significant software TCB) and a special physical network interface that can be used simultaneously by both the operating system and the management system. In contrast, CIDER implements dominance without requiring complex additional hardware and software.

A large body of work explores the definition, implementation and use of several trusted computing primitives, focusing on confidentiality and integrity. This includes secure and high-assurance boot [41], measured boot and remote attestation [12–15] and isolated or shielded execution [16–21]. One line of work aims to enable attestation with minimal hardware support [67–77]. Several dynamic attestation approaches based on creating isolated execution environments have been proposed [78–82], while allowing dynamic code loading [78, 82]. Security architectures for process sandboxing and memory isolation [83–85] or privilege separation [86, 87] have also been proposed for resource-constrained devices, mitigating the effect of software bugs and of the exploitation of vulnerabilities. This line of work, however, does not consider availability (or recoverability), mostly due to its potential complexity. In contrast, CIDER ensures availability using simple security primitives.

Work on secure code updates for resource-constrained devices comes closer to CIDER. Secure update mechanisms were proposed primarily for nodes in sensor networks and combined with attestation [88–93]. Work in this area aims to improve the update protocol by making it robust against broader classes of attacks [94], adapting it to the constraints of embedded and IoT devices [95, 96] and reducing its overhead [96]. Key security goals include reliably detecting and blacklisting devices on which updates fail to install [88–90, 92, 93], protecting the secrecy of the update [91] and guaranteeing that rogue programs disguised as updates will not be installed [94, 95]. CIDER is largely orthogonal to this line of work. CIDER is a system architecture that guarantees that an update mechanism will be invoked within a prescribed amount of time. In principle, CIDER could be used to force invocation of many existing update protocols.

Azure Sphere [97–99] and Android Things [100] are commercial IoT platforms with a variety of security features including secure boot, runtime isolation, attestation, hardware-based key protection and software updates. Azure Sphere has evolved from the Sopris system [99]. Sopris adds a hardware security subsystem, including a separate security processor, key storage, crypto accelerators and a TRNG to the MediaTek MT7687 MCU. The addition of custom security hardware into the MCU makes Sopris quite different from CIDER, as it constitutes a fundamentally different trade-off between compatibility with existing microcontrollers and several key aspects of system design. Android Things uses a stripped-down version of the Android operating system. Its update mechanism is capable of recovering from compromised or broken applications. However, there is no means of recovering from OS compromise or secure boot failure after, for example, a rootkit infection.

X. Conclusion

In this paper, we introduce *dominance*, a novel trusted computing primitive for IoT. From a system administrator's point of view, dominance extends remote attestation and brings manageability of IoT devices to a new level: beyond obtaining evidence that a device is in good condition, dominance enables the administrator to remotely dictate the software that runs on the devices he/she manages via the hub, regardless of whether the device firmware is out of control. Dominance is ensured as long as the hub is secure.

In search of a practical scheme, we decompose dominance into two components, gated boot and a reset trigger, where gated boot ensures that only software allowed by the administrator can be executed on the device, and the reset trigger ensures that control can always return to gated boot within a bounded amount of time at the discretion of the remote administrator.

We further identify a small set of hardware features required to realize these primitives: including latches and an AWDT. Latches are readily available on many devices and we show that the AWDT can be easily realized.

Based on these ingredients, we propose an end-to-end system: CIDER. CIDER features transparency in its design by introducing minimum disruption when the device is operating normally, and only brings visible effects (e.g., forcing the device to reset) when the device does not respond to the administrator's requests, or a firmware update is necessary.

We show that CIDER has the potential to be adopted by a broad class of IoT devices by developing fully-functional prototypes of CIDER on three popular IoT platforms: HummingBoard Edge, Raspberry Pi Compute Module 3 and Nucleo-L476RG. Our evaluation shows that the overhead of CIDER is minimal and that it works seamlessly with complex deployments, such as Windows 10 IoT Core, Debian, Raspbian, and with bare-metal applications on constrained devices.

XI. Acknowledgments

We would like to thank Jordan Rhee for his generous help. We also thank the reviewers for their helpful feedback. Manuel Huber was partly supported by the Fraunhofer Cluster of Excellence 'Cognitive Internet Technologies.'

References

[1] Ericsson, "IoT Market Outlook," 2018, https://www.ericsson.com/en/networks/trending/hot-topics/iot-connectivity/iot-market-outlook.

[2] Libelium, "Top 50 Internet of Things Applications," 2018, http://www.libelium.com/resources/top_50_iot_sensor_applications_ranking.

[3] Intel Corporation, "Intelligent Vending with Intel IoT Retail Gateway," 2018, https://www.intel.com/content/www/us/en/embedded/retail/vending/iot-gateway-for-intelligent-vending/overview.html.

[4] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, "SoK: Security Evaluation of Home-Based IoT Deployments," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[5] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, , and Y. Zhou, "Understanding the Mirai Botnet," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[6] O. Çetin, C. Gañán, L. Altena, T. Kasama, D. Inoue, K. Tamiya, Y. Tie, K. Yoshioka, and M. van Eeten, "Cleaning Up the Internet of Evil Things: Real-World Evidence on ISP and Consumer Efforts to Remove Mirai," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[7] S. Herwig, K. Harvey, G. Hughey, R. Roberts, and D. Levin, "Measurement and Analysis of Hajime, a Peer-to-peer IoT Botnet," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[8] S. Soltan, P. Mittal, and H. V. Poor, "BlackIoT: IoT Botnet of High Wattage Devices Can Disrupt the Power Grid," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 15–32. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/soltan

[9] C. Cimpanu, ""Hide and Seek" Becomes First IoT Botnet Capable of Surviving Device Reboots," 2018, https://www.bleepingcomputer.com/news/security/hide-and-seek-becomes-first-iot-botnet-capable-of-surviving-device-reboots/.

[10] Kenin, Simon, "BrickerBot mod_plaintext Analysis," 2017, hxxps://www.trustwave.com/Resources/SpiderLabs-Blog/BrickerBot-mod_plaintext-Analysis.

[11] O'Malley, Mike, "The 7 Craziest IoT Device Hacks," 2018, https://blog.radware.com/security/2018/05/7-craziest-iot-device-hacks.

[12] Trusted Computing Group, "TPM Main Specification Version 1.2 Rev. 116," 2011, https://trustedcomputinggroup.org/resource/tpm-main-specification/.

[13] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman, "A Trusted Open Platform," *Computer*, vol. 36, no. 7, pp. 55–62, Jul. 2003. [Online]. Available: http://dx.doi.org/10.1109/MC.2003.1212691

[14] T. Garfinkel, M. Rosenblum, and D. Boneh, "Flexible OS Support and Applications for Trusted Computing," in *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, ser. HOTOS'03. Berkeley, CA, USA: USENIX Association, 2003, pp. 25–25. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251054.1251079

[15] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and Implementation of a TCG-based Integrity Measurement Architecture," in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, ser. SSYM'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 16–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251375.1251391

[16] James Greene, "Intel Trusted Execution Technology: Whitepaper," Intel Corporation, Tech. Rep., 2012.

[17] D. Kaplan, J. Powell, and T. Woller, "AMD Memory Encryption," Advanced Micro Devices, Tech. Rep., 2016.

[18] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB Reduction and Attestation," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 143–158. [Online]. Available: http://dx.doi.org/10.1109/SP.2010.17

[19] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," in *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13. New York, NY, USA: ACM, 2013, pp. 10:1–10:1. [Online]. Available: http://doi.acm.org/10.1145/2487726.2488368

[20] V. Costan and S. Devadas, "Intel SGX Explained," *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.

[21] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An Execution Infrastructure for TCB Minimization," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, ser. Eurosys '08. New York, NY, USA: ACM, 2008, pp. 315–328. [Online]. Available: http://doi.acm.org/10.1145/1352592.1352625

[22] Intel Corporation, "Intelligent Platform Management Interface Specification v2.0 rev. 1.1," Intel Corporation, Hewlett-Packard, NEC, Dell, Tech. Rep., 2013.

[23] Intel Corporation, "What is Intel Management Engine?" 2017, https://www.intel.com/content/www/us/en/support/articles/000008927/software/chipset-software.html?productId=34227&localeCode=us_en.

[24] Intel LAN Access Division, "Intel Sideband Technology: An Overview of The Intel Server Manageability Interfaces," 2009, https://www.intel.com/content/dam/doc/application-note/sideband-technology-appl-note.pdf.

[25] P. Oester, "Dirty COW CVE-2016-5195," 2016, https://dirtycow.ninja.

[26] T. Claburn, "Intel finds critical holes in secret Management Engine hidden in tons of desktop, server chipsets," 2017, https://www.theregister.co.uk/2017/11/20/intel_flags_firmware_flaws/.

[27] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna, "Boomerang: Exploiting the Semantic Gap in Trusted Execution Environments," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

[28] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *40th IEEE Symposium on Security and Privacy (SP'19)*, 2019.

[29] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[30] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[31] SolidRun, "HummingBoard Gate/Edge," https://developer.solid-run.com/products/hummingboard-gate-edge/.

[32] Raspberry Pi, "Compute Module 3," https://www.raspberrypi.org/products/compute-module-3/.

[33] STMicroelectronics, "NUCLEO-L476RG - STM32 Nucleo-64 development board with STM32L476RG MCU, supports Arduino and ST morpho connectivity," https://www.st.com/en/evaluation-tools/nucleo-l476rg.html.

[34] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A Verified Modern Cryptographic Library," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[35] United States Environmental Protection Agency (EPA), "Air Data: Air Quality Data Collected at Outdoor Monitors Across the US," 2018, https://www.epa.gov/outdoor-air-quality-data.

[36] SierraWireless, "Smart Traffic Lights Help Ease the Burden of Rush Hour on City Infrastructure," 2017, https://www.sierrawireless.com/iot-blog/iot-blog/2017/07/smart-_traffic_lights_help_ease_the_burden_of_rush_hour_on_city_infrastructure.

[37] Slowey, Lynne, "KONE makes elevator services truly intelligent with Watson IoT," 2017, https://www.ibm.com/blogs/internet-of-things/kone.

[38] Trusted Computing Group, "Hardware Requirements for a Device Identifier Composition Engine," 2018.

[39] Symantec, "Symantec Internet Security Threat Report," *White paper, Symantec Enterprise Security*, vol. 21, 2016.

[40] Android Open Source Project, "Implementing dm-verity," 2018, https://source.android.com/security/verifiedboot/dm-verity.

[41] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A Secure and Reliable Bootstrap Architecture," in *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, ser. SP '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 65–. [Online]. Available: http://dl.acm.org/citation.cfm?id=882493.884371

[42] S. Schulz, A. Schaller, F. Kohnhäuser, and S. Katzenbeisser, "Boot Attestation: Secure Remote Reporting with Off-The-Shelf IoT Sensors," in *Computer Security – ESORICS 2017*. Cham: Springer International Publishing, 2017, pp. 437–455.

[43] Project Everest, "hacl-star/snapshots/hacl-c," https://github.com/project-everest/hacl-star/tree/master/snapshots/hacl-c.

[44] Microsoft Azure, "Functions," https://azure.microsoft.com/en-us/services/functions/.

[45] The Legion of the Bouncy Castle, "Bouncy Castle C# API," http://www.bouncycastle.org/csharp/.

[46] Microsoft Azure, "SQL Database," https://azure.microsoft.com/en-us/services/sql-database/.

[47] STMicroelectronics, "NUCLEO-L053R8 - STM32 Nucleo-64 development board with STM32L053R8 MCU, supports Arduino and ST morpho connectivity," https://www.st.com/en/evaluation-tools/nucleo-l053r8.html.

[48] Microchip Technology Inc., "ATECC608A: CryptoAuthentication device summary datasheet," http://ww1.microchip.com/downloads/en/DeviceDoc/ATECC608A-CryptoAuthentication-Device-Summary-Data-Sheet-DS40001977B.pdf.

[49] ——, "ATtiny212/412: AVR Microcontroller with Core Independent Peripherals and picoPower technology," http://ww1.microchip.com/downloads/en/DeviceDoc/40001911A.pdf.

[50] Linaro Limited, "Open Portable Trusted Execution Environment," 2018, https://www.op-tee.org/.

[51] DENX Software Engineering, "Das U-Boot – The Universal Boot Loader," http://www.denx.de/wiki/U-Boot.

[52] D. Hare, "Adding TCP and wget into u-boot," https://lists.denx.de/pipermail/u-boot/2018-March/323699.html.

[53] JEDEC Standard, "Embedded Multimedia Card (eMMC), Electrical Standard 5.1," *JEDEC Solid State Technology Association, JESD84-B51*, vol. B51, 2015.

[54] Microsoft, "Windows 10 Internet of Things," https://developer.microsoft.com/en-us/windows/iot.

[55] Software in the Public Interest, Inc., "Debian, the universal operating system," https://www.debian.org/.

[56] Raspberry Pi Documentation, "Compute Module and related schematics," https://www.raspberrypi.org/documentation/hardware/computemodule/schematics.md.

[57] Raspbian.org, "Raspbian," https://www.raspbian.org/.

[58] Buildroot.org, "Buildroot, making embedded Linux easy," https://buildroot.org/.

[59] STMicroelectronics, "STM32L4 - Firewall," https://www.st.com/.../product_training/stm32l4_security_firewall.pdf.

[60] Github, "NUCLEO-L476RG_DFSDM_PDM-Mic," 2017, https://github.com/y2kblog/NUCLEO-L476RG_DFSDM_PDM-Mic.

[61] ST-Link, "STM32CubeL4," 2017, https://www.st.com/content/st_com/en/products/embedded-software/mcus-embedded-software/stm32-embedded-software/stm32cube-mcu-packages/stm32cubel4.html.

[62] Standard Performance Evaluation Corporation, "The SPEC CPU 2006 Benchmark Suite," https://www.spec.org/.

[63] EEMBC Embedded Microprocessor Benchmark Consortium, "CoreMark - An EEMBC Benchmark," https://www.eembc.org/coremark/.

[64] Intel Corporation, "Data Center Manageability Interface Specification v1.5 rev. 1.0," Intel Corporation, Microsoft Global Foundation Services, Dell, Fujitsu, Hewlett-Packard, SGI, ZT Systems, Tech. Rep., 2011.

[65] Distributed Management Task Force, "Redfish Scalable Platforms Management API Specification v1.5," Redfish, Tech. Rep., 2018.

[66] Intel Corporation, "Intel Active Management Technology SDK," 2018, https://software.intel.com/en-us/amt-sdk.

[67] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, "SWATT: SoftWare-based ATTestation for Embedded Devices," in *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, May 2004, pp. 272–282.

[68] M. Jakobsson and K.-A. Johansson, "Retroactive Detection of Malware with Applications to Mobile Platforms," in *Proceedings of the 5th USENIX Conference on Hot Topics in Security*, ser. HotSec'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924931.1924941

[69] Y.-G. Choi, J. Kang, and D. Nyang, "Proactive Code Verification Protocol in Wireless Sensor Network," in *Proceedings of the 2007 International Conference on Computational Science and Its Applications - Volume Part II*, ser. ICCSA'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 1085–1096. [Online]. Available: http://dl.acm.org/citation.cfm?id=1802954.1803061

[70] Y. Yang, X. Wang, S. Zhu, and G. Cao, "Distributed Software-based Attestation for Node Compromise Detection in Sensor Networks," in *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, ser. SRDS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 219–230. [Online]. Available: http://dl.acm.org/citation.cfm?id=1308172.1308237

[71] M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim, "Remote Software-based Attestation for Wireless Sensors," in *Proceedings of the Second European Conference on Security and Privacy in Ad-Hoc and Sensor Networks*, ser. ESAS'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 27–41. [Online]. Available: http://dx.doi.org/10.1007/11601494_3

[72] Y. Li, J. M. McCune, and A. Perrig, "VIPER: Verifying the Integrity of PERipherals' Firmware," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 3–16. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046711

[73] F. Armknecht, A.-R. Sadeghi, S. Schulz, and C. Wachsmann, "A Security Framework for the Analysis and Design of Software Attestation," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516650

[74] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik, "A Minimalist Approach to Remote Attestation," in *Proceedings of the Conference on Design, Automation & Test in Europe*, ser. DATE '14. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2014, pp. 244:1–244:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616606.2616905

[75] P. England, R. Aigner, K. Kane, A. Marochko, D. Mattoon, R. Spiger, S. Thom, and G. Zaverucha, "Device Identity with DICE and RIoT: Keys and Certificates," Microsoft, Tech. Rep., September 2017. [Online]. Available: https://www.microsoft.com/en-us/research/publication/device-identity-dice-riot-keys-certificates/

[76] L. Jäger, R. Petri, and A. Fuchs, "Rolling DICE: Lightweight Remote Attestation for COTS IoT Hardware," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ser. ARES '17. New York, NY, USA: ACM, 2017, pp. 95:1–95:8. [Online]. Available: http://doi.acm.org/10.1145/3098954.3103165

[77] W. Feng, Y. Qin, S. Zhao, and D. Feng, "AAoT: Lightweight attestation and authentication of low-resource things in IoT and CPS," *Computer Networks*, vol. 134, pp. 167 – 182, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128618300471

[78] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust," in *NDSS*, vol. 12, 2012, pp. 1–15.

[79] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A Security Architecture for Tiny Embedded Devices," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: ACM, 2014, pp. 10:1–10:14. [Online]. Available: http://doi.acm.org/10.1145/2592798.2592824

[80] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 479–494. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534808

[81] N. Asokan, F. Brasser, A. Ibrahim, A.-R. Sadeghi, M. Schunter, G. Tsudik, and C. Wachsmann, "SEDA: Scalable Embedded Device Attestation," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 964–975. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813670

[82] F. Brasser, B. E. Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: Tiny Trust Anchor for Tiny Devices," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

[83] ARM Limited, "mbed OS," 2018, https://www.mbed.com/en/platform/mbed-os.

[84] Z. B. Aweke and T. M. Austin, "uSFI: Ultra-Lightweight Software Fault Isolation for IoT-Class Devices," *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1015–1020, 2018.

[85] R. Strackx, F. Piessens, and B. Preneel, "Efficient Isolation of Trusted Subsystems in Embedded Systems," in *Security and Privacy in*

Communication Networks, S. Jajodia and J. Zhou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 344–361.

[86] C. H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, and D. Xu, "Securing Real-Time Microcontroller Systems through Customized Memory View Switching," in *Network and Distributed Systems Security Symp.(NDSS)*, 2018.

[87] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer, "Protecting Bare-Metal Embedded Systems with Privilege Overlays," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 289–303.

[88] A. Seshadri, M. Luk, A. Perrig, L. v. Doorn, and P. Khosla, "Using FIRE & ICE for Detecting and Recovering Compromised Nodes in Sensor Networks," School of Computer Science, Carnegie-Mellon University, Tech. Rep., 2004.

[89] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla, "SCUBA: Secure Code Update By Attestation in Sensor Networks," in *Proceedings of the 5th ACM Workshop on Wireless Security*, ser. WiSe '06. New York, NY, USA: ACM, 2006, pp. 85–94. [Online]. Available: http://doi.acm.org/10.1145/1161289.1161306

[90] D. Perito and G. Tsudik, "Secure Code Update for Embedded Devices via Proofs of Secure Erasure," in *Proceedings of the 15th European Conference on Research in Computer Security*, ser. ESORICS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 643–662. [Online]. Available: http://dl.acm.org/citation.cfm?id=1888881.1888931

[91] C. Huth, P. Duplys, and T. Güneysu, "Secure Software Update and IP Protection for Untrusted Devices in the Internet of Things via Physically Unclonable Functions," in *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, March 2016, pp. 1–6.

[92] F. Kohnhäuser and S. Katzenbeisser, "Secure Code Updates for Mesh Networked Commodity Low-End Embedded Devices," in *Computer Security – ESORICS 2016*, I. Askoxylakis, S. Ioannidis, S. Katsikas, and C. Meadows, Eds. Cham: Springer International Publishing, 2016, pp. 320–338.

[93] W. Feng, Y. Qin, S. Zhao, Z. Liu, X. Chu, and D. Feng, "Secure Code Updates for Smart Embedded Devices based on PUFs," Cryptology ePrint Archive, Report 2017/991, 2017, https://eprint.iacr.org/2017/991.

[94] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine, "Survivable Key Compromise in Software Update Systems," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866315

[95] 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.

[96] N. Asokan, T. Nyman, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik, "ASSURED: Architecture for Secure Software Update of Realistic Embedded Devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems - Special Issue ESWEEK 2018, CASES 2018, CODES + ISSS 2018 and EMSOFT 2018*, July 2018. [Online]. Available: http://tubiblio.ulb.tu-darmstadt.de/105801/

[97] G. Hunt, "Introducing Microsoft Azure Sphere: Secure and power the intelligent edge," 2018, https://azure.microsoft.com/en-us/blog/introducing-microsoft-azure-sphere-secure-and-power-the-intelligent-edge/.

[98] Microsoft, "Azure Sphere Documentation – What is Azure Sphere?" 2018, https://docs.microsoft.com/en-us/azure-sphere/product-overview/what-is.

[99] G. Hunt, G. Letey, and E. Nightingale, "The Seven Properties of Highly Secure Devices," Microsoft, Tech. Rep., March 2017. [Online]. Available: https://www.microsoft.com/en-us/research/publication/seven-properties-highly-secure-devices/

[100] Google, "Android Developers: Android Things," 2018, https://developer.android.com/things/.