Veto: Prohibit Outdated Edge System Software from Booting

Jonas Röckl[1], Adam Wagenhäuser[1] and Tilo Müller[2]
[1]*FAU Erlangen-Nürnberg, Germany*
[2]*Hof University of Applied Sciences, Germany*

Keywords: Trusted Execution Environment, Full Disk Encryption, Remote Attestation, Edge Computing.

Abstract: Edge computing emerges as a trend, forming a link between the Internet of Things and cloud-based services. Large-scale edge deployments are already in place today in the context of communication network providers that offload more and more tasks to the edge to ensure high flexibility and low latencies. By relying on remote attestation and disk encryption techniques, we design a novel system architecture that protects confidential data on edge nodes in the case of device theft. Recent vulnerabilities like *Ripple20* and *Amnesia:33* show the consequences and costs of critical security bugs stemming from outdated system software. Thus, we design our system in a way that a node can derive its decryption key if and only if a trusted remote party (e.g., a network operator) can verify that it is running the latest software. This is a security feature that prevalent implementations like Linux's dm-crypt lack. To secure the early-boot communication, we rely on a trusted execution environment, hardware offloading, and Rust device drivers. We prototype our system on two recent ARMv8 devices and show that the performance overhead ($\approx 2\%$) and the boot delay ($1s$) are low. Thus, we believe that our concept is a meaningful step towards more secure future edge devices.

1 INTRODUCTION

Driven by broad applications like home automation, energy metering, health monitoring, and large-scale industrial systems, the Internet of Things (IoT) grows every day. The constantly increasing demand for high bandwidth, low latencies, and strong privacy fosters location-aware edge computing models, forming a link between IoT devices and a cloud service. Edge computing differs from traditional cloud computing in the sense that it performs computing at the edge of the network (e.g., cell towers or distributed "micro" data centers), closer to the source of the data like IoT devices (Cao et al., 2020). Large edge computing deployments are in place already today in the context of communication network providers that deploy more and more intelligent network elements and offload a variety of tasks to the edge of the network. Edge nodes process and also store potentially sensitive data from a growing number of stakeholders (Röckl et al., 2021). In the future, a wide variety of data is expected to be processed on the edge, including but not limited to data from health monitoring systems, GPS data, CCTV surveillance streams, autonomous driving systems, and multiple more.

In contrast to highly-protected cloud data centers, physical attacks on edge nodes are a realistic scenario because access cannot be limited to authorized personnel only due to geographic dispersion (Busch et al., 2019). This results in attack vectors to network infrastructure like device theft, threatening the data on edge devices.

The recently published sets of vulnerabilities named *Ripple20* and *Amnesia:33* expose a plethora of critical bugs in widely-used network stacks, with some of them enabling remote code execution (Kol and Oberman, 2020; Forescout Research Labs, 2020). For example, CVE-2020-11901 allows remote code execution with a forged DNS response. Those incidents clearly show the consequences and costs of critical vulnerabilities in outdated system software. Most strikingly, some of the affected devices do not include an update mechanism to patch the vulnerabilities, exposing the device and, thus, potentially confidential data to an adversary indefinitely.

While there is research on disk encryption for user-facing devices like smartphones (Groß et al., 2021), Full Disk Encryption (FDE) for edge nodes is rarely dealt with. One challenge is that edge devices need to boot without actively entering a password during boot. To tackle that challenge, existing techniques like Microsoft's Bitlocker, Apple's File-

Röckl, J., Wagenhäuser, A. and Müller, T.
Veto: Prohibit Outdated Edge System Software from Booting.
DOI: 10.5220/0011627700003405
In *Proceedings of the 9th International Conference on Information Systems Security and Privacy (ICISSP 2023)*, pages 46-57
ISBN: 978-989-758-624-8; ISSN: 2184-4356

Vault, and Linux's dm-crypt can rely on a secret protected by hardware (e.g., by a TPM). Still, an attacker can steal the device and indefinitely boot the device. With the stolen device, the attacker has enough time to try to break into the system via software (jailbreak) or hardware vulnerabilities and to exfiltrate confidential data. For example, the attacker might be able to exploit a bug in the booted Operating System (OS) via a logical communication channel (e.g., a serial console) that was not or not sufficiently secured. Even if the vulnerabilities are fixed by the vendor, there is no way to ensure that the updates are installed on the stolen device. Thus, novel anti-theft techniques for edge infrastructure are to be developed.

Contributions. We design a novel system architecture that protects confidential data on edge nodes in the case of device theft. A trusted remote party (e.g., a network operator) can actively decide if a device with a given software state is allowed to boot the OS.

If the trusted remote party deems the device secure (e.g., by verifying that all recent software updates are installed), the trusted party sends a token that is required to derive the key for the FDE. This way, outdated and potentially vulnerable system software is not able to derive the FDE key, mitigating data breaches on stolen edge computing nodes. In summary, we make the following contributions:

- We design a novel system architecture that binds booting the OS on edge nodes to the most up-to-date system software.
- We develop a protocol to securely retrieve a token from a trusted party during the boot process, which is used to derive the FDE key.
- We propose and implement several strategies to isolate an early-boot communication channel from system resources including key material.
- We prototype our system on two ARMv8-A devices. We evaluate the size of the Trusted Computing Base (TCB) and discuss security.
- We measure the CPU overhead and the boot delay.

2 THREAT MODEL

In our work, we focus on the confidentiality of the data on edge devices. We require that the communication interfaces of the device (e.g., PCI, serial console, and JTAG) are either disabled or protected via appropriate authentication mechanisms. We assume that these mechanisms are strong enough to hinder the adversary from conducting runtime attacks directly at the site of deployment (e.g., brute-forcing the credentials for a local serial console). We instead focus on an attacker that steals the device and conducts *boot-time* attacks. We identify threats ($\mathfrak{T}s$). In the attacker-controlled environment, the adversary can read ($\mathfrak{T}1$) and write blocks ($\mathfrak{T}2$) by using the storage device's interfaces (e.g., a SATA cable or the exposed pins of an eMMC). The attacker can also mount downgrade attacks ($\mathfrak{T}3$) to the storage. Moreover, the adversary can conduct cold-boot attacks on external RAM as well as Direct Memory Access (DMA) attacks ($\mathfrak{T}4$). The adversary can emulate network traffic from and to the device ($\mathfrak{T}5$) and can connect additional peripherals ($\mathfrak{T}6$). The attacker may also exploit bugs in early-boot software ($\mathfrak{T}7$) akin to jailbreaks (axi0mX, 2019).

However, we assume that the attacker can neither access on-chip RAM nor extract hardware-protected secrets by relying on, for example, focused ion beams (Selmke et al., 2016). These attacks are expected to be highly intrusive, dependent on the hardware, and costly. Hardware side-channel attacks are out of scope and we presume that the cryptography primitives are secure. We assume that the hardware works according to the corresponding specification. This includes the eMMC Replay-Protected Memory Block (RPMB) which must never allow access without a valid Message Authentication Code (MAC).

Securing the trusted remote party is out of scope. We assume a cloud-based server and, thus, a wide range of existing protection techniques are available.

3 ARM TrustZone

The ARM TrustZone is a Trusted Execution Environment (TEE) available on most modern ARMv8-A CPUs. Two isolated computing domains, the Normal World (NW) and the Secure World (SW) are introduced (Pinto and Santos, 2019). The NW is also referred to as Rich Execution Environment (REE). A CPU status bit, the so-called Non-Secure (NS) bit, encodes the world a core currently executes in. The SW can access resources from the NW but not vice versa. The NS bit is passed down to the RAM and the peripheral buses. A dedicated peripheral, the TrustZone Protection Controller (TZPC) assigns memory-mapped peripherals to the SW or the NW. The TrustZone Address Space Controller (TZASC) allows partitioning DRAM into secure and non-secure partitions.

The hardware privilege levels of the CPU are called Exception Levels (ELs). EL0 is typically used for applications, EL1 for an OS, and EL2 provides hardware support for virtualization. Whereas EL0-EL2 exist in both worlds, EL3, the level with the highest privilege, only exists in the SW. The so-called *secure monitor* runs in EL3, which manages

context switches from and to the SW (ARM Limited, a), coordinates the power state of the CPU, and handles firmware updates (ARM Limited, b). Moreover, it includes a True Random Number Generator (TRNG) and a trusted boot process with a Chain of Trust (CoT). Typically, the Root of Trust (RoT) is stemmed from hardware. The secure monitor runs on on-chip memory to mitigate cold-boot attacks and DMA attacks.

The ARM TrustZone with a software stack like OP-TEE (Linaro Limited, 2022) does neither provide full disk encryption nor does it allow early-boot remote attestation for autonomously booting systems.

4 DESIGN

Fig. 1 illustrates the boot process of an edge device. The underlined components are extensions to the original boot flow that are gradually explained in this section. On ARMv8-A, the device boots up in the TEE. Typically, some vendor-provided software in a Read-Only Memory (ROM) runs after a device reset (1). The boot ROM loads and executes some sort of firmware (1). Any component after the boot ROM is signed and only executed after verifying the signature (secure boot). The signatures can be stored alongside the components. The hardware is required to support an RoT that allows storing a public key to verify the signatures. The public key must be protected from modifications. For this reason, some devices support blowing eFuses to pin the public key. The corresponding private key remains at the trusted remote party.

On a system *without* our proposed modifications, the firmware directly passes control to the REE (8) and loads the kernel or a second-stage boot loader. A kernel binary can include a small file system (*initfs*), which can set up dm-crypt to decrypt the actual root file system (*rootfs*). The REE's root file system is encrypted with an FDE key. On a conventional system, this key is derived from a passphrase that needs to be manually entered before the system boots.

We, however, carefully alter the boot process so that the device receives a token from a trusted party during the boot process which is used to derive the FDE key. We add the following lightweight components as an extension to the firmware:

Governor. The governor is responsible for the coordination of the altered boot process. It is called by the firmware (2) and returns to it after execution (7). The governor supervises the retrieval of a token from the trusted remote party and derives the FDE key. The REE can only boot if the rootfs is decrypted. This depends on deriving the FDE key, which is only possible with the token. Subsidiaries to the governor are the emergency switch, the confinement module, and the agent. They are dealt with in the following.

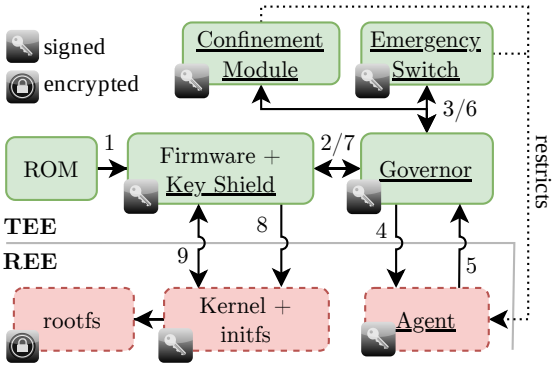

Figure 1: The boot process. Green components with solid border are trusted. Red components with dashed border are untrusted. Underlined components extend the original boot process. The numbers show the boot order.

Emergency Switch. A security functionality that prevents an attack on the agent from indefinitely monopolizing the device. The emergency switch ensures that the control flow is passed back to the governor in the TEE even if the system behaves unexpectedly. One option to implement the emergency switch is a hardware timer interrupt. Likewise, hardware watchdogs are widely available (Xu et al., 2019). As a requirement, the interrupt may not be prevented by the agent. Modern TEEs can assign the interrupt or the watchdog to the TEE only.

Confinement Module. While the emergency switch isolates the agent in a temporal fashion, the confinement module does so in a spatial manner. The confinement module uses hardware-based protection techniques to securely restrict the agent to certain regions of memory and peripherals.

Agent. The governor activates the emergency switch and the confinement module (3) before the agent is called (4). This is because the agent is exposed to a network-facing attacker and, thus, is considered untrusted. The agent communicates with the trusted party to receive the token. To do so, the agent sets up a network connection. In this work, we rely on an external hardware network stack connected to the edge device (Sec. 5). Subsequently, the agent submits the token to the governor (5). The governor disables the confinement module and the emergency switch again (6). The governor can access a local device secret derived from the RoT of the device. Based on the local device secret and the token, the governor derives an FDE key. Finally, the governor passes the FDE key to the key shield.

Key Shield. In addition to boot flow modifications, we add a runtime component called the key shield to

the firmware. We refer to it as a *runtime* component, since the REE software can use it after it has booted. The key shield is similar to a trusted key store (Android, 2022). In contrast, one cannot dynamically add and remove keys. During the boot process, the key shield receives the FDE key from the governor. After initialization, the REE can ask the key shield to conduct encryption and decryption with this key (9). Yet, the key is never directly exposed to the REE.

Trusted Remote Party. Following the edge computing model (Cao et al., 2020), we assume a trusted remote party (e.g., the network operator) to operate a cloud-based server that decides whether an edge device with an attested software state is allowed to decrypt the rootfs and boot the REE.

4.1 Remote Attestation

We use static remote attestation similar to related works (Röckl et al., 2021; Huber et al., 2020; Xu et al., 2019). The firmware calculates an attestation value $A = H(M_1||M_2||...||M_n)$ where M_1, M_2, ... M_n are measurements of the early-boot software components. The attestation value covers the firmware with the key shield, the governor, the emergency switch, the confinement module, the agent, and the kernel binary including the initfs. We attest the state to prevent outdated system software from successfully receiving the token and booting the REE. We do not attest the state of the encrypted file system during *early boot*. If the use case requires it, one can rely on existing kernel features like dm-verity to attest the file system state *after* one has booted the REE kernel.

4.2 Key Derivation Protocol

We design a lightweight protocol so that a device D can retrieve a token T from a trusted remote party P. The token T is used to derive an FDE key. The design goals for the protocol are as follows: The protocol is required to ① ensure the confidentiality of T, ② check the authenticity of D and P, and ③ be resistant to replay attacks. Moreover, ④ the protocol needs to last one Round Trip Time (RTT). This is because the governor can then prepare a request and pass it to the agent. Afterward, the agent returns the response to the governor for verification. This way, no cryptography is needed in the REE. Finally, ⑤ the protocol needs to add a low amount of code to the TCB. We introduce the following notations. A key pair (K_Y, K_Y^{-1}) of entity Y consists of a public key K_Y and a private key K_Y^{-1}. $M_{K_Y^{-1}}$ is a message M, signed by Y.

Based on a hardware device secret X, D derives $X_T = H(X||\text{“}token\text{”})$ and $X_C = H(X||\text{“}counter\text{”})$. Moreover, D derives a key pair (K_D, K_D^{-1}) from the value $H(X||\text{“}cert\text{”})$. P has a unique key pair (K_P, K_P^{-1}) for each device to communicate with. During provisioning, which is assumed to take part in a safe environment, D gets to know K_P. In return, P receives K_D and X_T.

When deployed, the device conducts the following steps during the boot process:

1. D sends a request $req = (A, C, N)_{K_D^{-1}}$. The term A is the attestation value that statically remote attests the state of D (Sec. 4.1). C is a non-volatile boot counter. We require that C can only be incremented if one knows X_C. We use an eMMC RPMB (Western Digital, 2017), which is a hardware-protected partition that only allows authenticated reads and writes. We set the RPMB key to X_C. One can do this only once. N is a nonce, randomly generated for each *req*.
2. P verifies the signature of *req* and decides on whether D is allowed to decrypt the disk and, thus, boot the REE. P can base the decision on the software state A. Furthermore, P can take the device's behavior into account, depending on the specific use case. For example, rolling restarts (as indicated by C) might be a sign of tampering. Similarly, a validly signed request from an unexpected source IP address may indicate a stolen device. We note that the actual detection of theft could be improved via sensors and means of surveillance. This is a research field orthogonal to our system.
3. If a device theft (or another form of irregular behavior) is detected, P does not hand out the token any longer. The communication aborts at this point. If the software state A is deprecated and, thus, an update mechanism is to be triggered, P answers with an error $err = (C, N, \text{“}deprecated\text{”})_{K_P^{-1}}$. To permit the boot, P derives a symmetric key K_T from the secret X_T. Then, P responds with $resp = (C, N, E)_{K_P^{-1}}$. The encrypted payload $E = (T)_{K_T}$ consists of the token T, symmetrically encrypted with K_T.
4. D receives *resp* or *err* and verifies both the signature and the correctness of C and N. In the case of *err*, an update mechanism is triggered (Sec. 4.3) and the protocol concludes at this point. In the case of *resp*, D derives K_T and decrypts E to retrieve the token T.
5. D derives the FDE key as $K = H(T||L)$ where $L = H(X||\text{“}fde\text{”})$. The latter binds the FDE key to the device which is also referred to as *sealing*.

We sketch reasoning on why the protocol fulfills its design goals. T is encrypted by K_T, which is derived

from X_T. Only P and D know X_T. On D, X_T and X never leave the TEE, and P is trusted. Thus, ① holds. Every message is signed by K_P^{-1} or K_D^{-1}. The private keys are only known to P and D. On D, K_D^{-1} never leaves the TEE, nor does X. Thus, ② holds.

In terms of replay attacks, one needs to distinguish between (a) injecting a previous or foreign message to a protocol run and (b) delaying a message, i.e., the intended party receives the unaltered message later than usual (Syverson, 1994).

We first analyze (a). Messages from or to one device cannot be replayed to another because each device has a unique key pair and P has a unique key pair for each device. Moreover, replaying a *req* from a former boot cycle of the device to P is not possible. This is because P stores the last value of C. A *req* with a C lower than the remembered is dropped. While replaying a *req* from the current boot cycle to P is possible technically, the net effect does not change. P has already decided on a *req* with these particular values of A and C in the past and returns the same *resp* or *err*. Replaying *resp* or *err* to D is not possible. This is because the internal value of N changes as soon as a response is received. A replayed *resp* has a deprecated N and does not pass the check in the governor.

Delaying a message (b) includes intercepting, repressing the original communication, and sending the unaltered message later on. We distinguish between an attack that involves device theft and one without. Delaying a *req* or *resp* without a device theft can lead to a Denial of Service (DoS) which we deal with in Sec. 4.4. Including device theft, an attacker can intercept a *req* and suppress further communication. Subsequently, the adversary steals the device and forwards *req* to P in the attacker-controlled environment, trying to receive a valid *resp*. However, this is only possible if the governor's internal values of C and N do not change during the theft. In other words, the device must not reboot while being stolen. If it does, however, C is increased and N is regenerated. A *resp* with a lower value of C or a differing nonce is not accepted. This work focuses on boot-time attacks (Sec. 2). Stealing a device without rebooting is seen as a runtime attack, which is out of scope. In light of this analysis, we conclude that the protocol is resistant to replay attacks (③).

An integer wrap-around of C allows attacking the protocol. The adversary steals the device and reboots it until the counter equals the value that was last accepted by P. Subsequently, D sends a valid *req*, answered with a valid *resp*. Since the counter is only increased during a reboot, the increment is tied to a certain duration. We propose to use a counter of a size so that rebooting the device until the counter wraps around takes longer than the life span of the device (e.g., hundreds of years). In Sec. 5, we show that a 64-bit counter is sufficient in our case.

We rely on the shared secret X_T and symmetric cryptography to encrypt T. This is because we want to keep the TCB small. The key shield already gives us symmetric encryption. Further, an asymmetric *signature* algorithm is required. Generally speaking, one cannot use asymmetric encryption and signatures interchangeably. To prevent also adding asymmetric *encryption*, we use the symmetric encryption already in the TCB. In Sec. 6, we quantify the TCB.

One alternative is not to rely on a token but to derive the FDE key from a device secret only. We deliberately refrain from this option. This is because we rather base the confidentiality guarantees on cryptography only instead of the whole TCB incl. cryptography. If P decides to prohibit the boot of D from a point in time onwards (e.g., because a device theft was detected), even vulnerabilities in the TCB (excl. cryptography) do not threaten confidentiality. This is because the required information to derive the FDE key is simply not physically available on the device.

4.3 Software Update

The trusted party can answer with *err*, indicating that the software in state A is deprecated (Sec. 4.2). To update the early-boot system software, one can rely on an existing early-boot update mechanism. These are dealt with extensively in related literature and do not require a booted OS (Xu et al., 2019; Huber et al., 2020; Röckl et al., 2021). Typically, the works implement a two-phase approach. First, an early-boot network stack retrieves a software update image. During the execution of the networking stack, other system components need to be protected. For example, Xu et al. rely on storage write protection to limit the consequences of early-boot breaches (Xu et al., 2019), while we propose to use a hardware network stack, TEE isolation techniques, and memory-safe and type-safe device drivers to secure the agent (Sec. 5). After the download, a dedicated component verifies and installs the update image. They also include some sort of update trigger to force the reset of fully-booted devices, limiting the time the device can run on deprecated system software. We carefully design our architecture in a way that the mentioned existing early-boot update mechanisms can be integrated.

4.4 Denial of Service

D may not receive a response from the trusted party if the communication between D and P is disrupted. In

that case, *D* tries to receive a response from *P* periodically with some reasonable frequency depending on the use case (e.g., once every minute). If a device theft has been detected, *P* rejects further boot processes of *D* by aborting the communication.

This might enable a DoS attack for a network-facing attacker. By hindering the communication between *D* and *P*, the device cannot receive the token and, thus, cannot boot the REE. While we recognize that the protocol is not suited for every deployment, we argue that this is justifiable in the domain of large-scale edge computing. This is for the following reasons. First, we assume that long-standing network-level attacks and outages can be mitigated directly at the infrastructure level. This is a valid strategy already established widely in the literature (Xu et al., 2019; Huber et al., 2020; Röckl et al., 2021; Suzaki et al., 2020). Being close to the network edge (or even *at* the network's edge) fosters a global view of the traffic and, thus, enables targeted (D)DoS mitigation already available today (Cloudflare Inc., 2022).

In addition to that, the edge computing paradigm inherently introduces redundancy, mitigating (D)DoS attacks to the complete system. The edge computing paradigm relies on failover techniques to ensure that another device takes over temporarily if one edge device fails (Yu et al., 2018; Harchol et al., 2020). Symantec discovered that less than one percent of the network-based (D)DoS attacks last more than 24 hours (Symantec, 2016; Xu et al., 2019). In light of this, we deem a temporary failover plausible.

Due to targeted (D)DoS mitigation techniques at the edge of the network and the inherent redundancy of the edge computing model, we argue that it is unrealistic that an attacker is able to attack a significant amount of the available edge devices in large deployments simultaneously.

5 IMPLEMENTATION

We prototype our approach on a Nitrogen8M development board from Boundary Devices with an ARMv8-A Cortex-A53 CPU, 2GB of external RAM, an 8GB eMMC with RPMB, and a secure boot implementation based on eFuses. Due to a missing expansion card (chip shortage) for the Nitrogen8M, we have not been able to implement the agent on the Nitrogen8M. Thus, the agent (Sec. 4) is implemented on a Raspberry Pi 4B with an ARMv8-A Cortex-A72 CPU and 8GB of RAM. With the expansion card, porting the agent is a pure engineering effort. We use Trusted Firmware-A (TFA), version 2.2, as an EL3 firmware. TFA is a widely deployed reference implementation. We rely on Linux in the NW. The emergency switch is implemented with a secure hardware watchdog. Currently, we do not yet use a hardware-protected device secret (*X*, Sec. 4.2) but rely on a hard-coded value. Yet, a unique chip identifier in an eFuse is available in hardware on the Nitrogen8M (NXP Semiconductors, 2022).

Implementing the Governor. We implement the governor in C as part of TFA. The confinement module and the emergency switch are object files linked to the governor. We port the Ed25519 asymmetric signature algorithm implementation from *HACL**, a formally verified cryptography library, to TFA (Zinzindohoué et al., 2017). The Nitrogen8M supports the ARMv8-A instruction set extensions for AES operations. We port the Linux drivers, mostly written in assembler, for the ARMv8-A AES hardware extensions to TFA on the Nitrogen8M.

We modify U-Boot SPL, a small component running after the boot ROM and before the TFA on the Nitrogen8M, to read and increase a counter (*C*, Sec. 4.2) on eMMC RPMB storage with every boot. As U-Boot SPL uses the eMMC to load the subsequent boot stages, only small additions for the RPMB are required. We use the existing RPMB code in U-Boot and link it to U-Boot SPL.

The counter *C* is an unsigned 64-bit value. To verify that the system is resistant to practical wrap-around attacks (Sec. 4.2), we evaluate the time the system needs from device reset to incrementing the counter. We use the system timer (`CNTPCT_EL0`) to quantify the execution time of U-Boot SPL and measure the interval from the initialization of the console to the increment of the boot counter. This is a lower bound since neither the boot ROM nor the time before console initialization is considered. Rebooting the device 16 times, we measure an average duration of $0.40492s$. Thus, a complete wrap-around takes at least $23.7 \cdot 10^{10}$ years, which is practically infeasible.

Implementing the Key Shield. The key shield is implemented as a *service* in TFA, which offers decryption and encryption to the NW via Secure Monitor Calls (SMCs). We rely on minimal-intrusive changes to the existing AES-XTS implementation in the Linux kernel. Whenever a so-called *alias key* is used for an AES-XTS operation, we do not execute the default Linux version, but trigger a context switch to the key shield via an SMC. The key shield operates with the actual key and returns to the NW. The alias key is not a secret. This way, we can re-use existing interfaces, i.e., dm-crypt and LUKS, without further modifications. On the contrary, we break cryptography if the alias key equals the actual key by coincidence. We consider the risk as neglectable in practice.

Implementing the Confinement Module. The agent is exposed to the NW and, thus, considered untrusted. Without any further measures, a compromised agent can take over the NW and interact with every non-secure peripheral. To prevent this, we grant the agent access to as few system resources and peripherals as possible. TFA already initializes the hardware to run a bare-metal application in the NW. Before the agent runs, the confinement module configures the TZPC (Sec. 3) to deny the NW from accessing any peripheral except a hardware network stack. Moreover, the TZASC restricts the agent to its region in NW RAM. When the agent returns, the previous configuration is restored. We implement the confinement module in C as an extension to TFA.

Implementing the Agent. The confinement module isolates the agent from the TEE. This section deals with the strategies on how to reduce the attack surface from the agent's perspective.

Network Stack Hardware Offloading. Software network stacks are complex and error-prone (Kol and Oberman, 2020; Forescout Research Labs, 2020). To counter, we deploy a Wiznet W5500 Ethernet shield as a hardware network stack, connected to the Raspberry Pi via Serial Peripheral Interface (SPI). These connect Ethernet to a significantly less complex logical communication protocol. While for low-end embedded devices ($3-10) the cost of such a peripheral might be higher than for the device itself, we argue that for edge devices ($100+) such a peripheral is reasonable economically. Importantly, they already provide on-chip implementations of IP, UDP, and TCP. This means that the network state machines, header parsing, and fragmentation logic are implemented on the peripheral. The SPI interface does not carry packages or datagrams, but payloads only (and some comparably simple control messages). We use UDP. This way, the peripheral does not need to maintain a TCP state internally, narrowing down the attack surface. If a network message is dropped, we just resend it after some time. The hardware network stack adds another level of isolation to the system. Even if the peripheral is vulnerable, an attacker still needs to expand their privileges to the edge device.

Memory-Safe and Type-Safe Agent. To impede the attacker from expanding their privileges from a potentially vulnerable network stack to the device, the agent is a bare-metal application written in pure Rust. Besides the application logic, the agent contains stripped-down drivers for the SPI controller, the GPIO controller, and the Ethernet shield. We need GPIO to configure and enable the SPI pins initially. A small subset of the drivers cannot be realized in a memory-safe fashion. This is because raw pointer access is necessary to communicate with memory-mapped peripherals. Still, the remaining parts can profit from memory-safe and type-safe programming, eradicating memory-corruption vulnerabilities. We use a custom library to wrap memory-mapped SPI controller registers to type-safe and memory-safe abstractions. Before the agent starts, the governor stores *req* in NW RAM. Subsequently, the agent transmits *req* to the trusted party and receives *resp*. The agent stores *resp* in NW RAM and returns it to the governor via an SMC. We restrict the SPI communication via pattern matching with the Rust `match` keyword to those control messages and payloads that are required. Any other message is denied. Because of the simple SPI protocol (especially when compared to IP), we argue that this makes it more difficult for an adversary to expand the privileges from the peripheral to the device.

Table 1: Evaluation of the TCB Size.

Software Component	LoC [N]	Binary [B]	Relative [%]
U-Boot SPL	46 884	64 470	70.95
Trusted Firmware-A	12 374	98 008	18.73
RPMB Driver	354	3226	0.54
Non-Volatile Boot Counter	44	702	0.07
AES-XTS Crypto Ext.	402	3102	0.61
Key Shield	140	794	0.21
Confinement Module	30	220	0.05
Emergency Switch	68	569	0.10
HACL* Ed25519	5609	30 112	8.49
Governor	173	1320	0.26
Total	66 078	178 213	100.00

6 SECURITY DISCUSSION

We evaluate the size of TCB and discuss the security.

6.1 Trusted Computing Base

The components of the TCB are listed in Tab. 1. As a baseline (first block), we choose U-Boot SPL and TFA which are the default software for the Nitrogen8M. For determining the Lines of Code (LoC), we first compile the software with debug symbols. Using `readelf` (version 2.34), we retrieve the contents of the `.debug_line` section. This way, we can get the name of any source code file that resulted in code in the binary. We use `cloc` (version 1.82) to count the LoC of the files. For highly-configurable multi-platform software like U-Boot and TFA, we deem that this is a much more appropriate estimation of the lines in the TCB than counting the LoC of all files. For determining the binary size, we use `size` (version 2.34). To quantify the additions to the TCB (second block), we use `cloc` on the source files of the added compo-

nent. Moreover, we use `size` on the added object file as an approximation for the binary additions of a component. The last row lists the LoC and the binary size including all extensions. The last column calculates the relative share based on LoCs.

Although the size of Ed25519 is not neglectable, HACL* is formally verified for correctness and memory safety and is resistant to some types of side-channel attacks. Therefore, we argue that an expansion of the TCB does not lead to a higher security risk in that case. In summary, we only add 1211 (1.83%) unverified LoC and 5609 (8.49%) formally verified LoC to the TCB.

6.2 Attacks and Countermeasures

We summarize potential attacks on our system.

Reading Data ($\mathfrak{T}1$). By external access, an attacker can read out the eMMC. She can access the unencrypted binaries of the boot components, i.e., the boot ROM, the firmware, the governor, the confinement module, the emergency switch, the agent, and the kernel binary including the initfs. This does not pose a threat since no secrets are at risk. She cannot read the boot counter from the RPMB. This requires X_C (Sec. 4.2) which we assume to be stored in secure TEE RAM. While she can access the encrypted rootfs, she cannot decrypt it. The key is either not physically available on the device or stored in on-chip TEE RAM. The device secret X (Sec. 4.2) is stored by specialized hardware (e.g., eFuses), which we assume to withstand an attacker. During runtime, X and its derivatives like X_C are only stored in secure RAM. Thus, the attacker is not able to derive the encryption key and no confidential data is leaked.

Writing Data ($\mathfrak{T}2$). Secure boot detects modifications to the early-boot software on eMMC storage. According to the threat model, changing the RoT is not possible because it is physically burnt to the device. While AES XTS is, theoretically, susceptible to manipulations of the ciphertext, one can additionally rely on integrity measures like *dm-verity* or file systems with checksums (e.g., ZFS). Thus, overwriting the ciphertext of the rootfs is not promising. Moreover, the attacker is not able to forge boot counter values on the RPMB. This is because writing requires X_C, which never leaves the on-chip TEE RAM.

Downgrade ($\mathfrak{T}3$). Downgrading the early-boot software to a previous but legitimate state is detected. This is because we include the attestation value A in our key derivation protocol (Sec. 4.2). As a consequence, the trusted party can hinder the REE from booting. Downgrade attacks to the boot counter are not possible. We require the hardware to work as specified (Sec. 2). Thus, one cannot change the boot counter on the RPMB without knowing X_C. Furthermore, replaying previous but legitimate requests to the RPMB controller is not promising. This is because every write request and read request also depends on the current state of the RPMB (Western Digital, 2017). The latter is important since one could rely on an eMMC sniffer (Tim Hummel, 2017) to maliciously modify the boot counter otherwise. This would allow setting the boot counter to the last value that caused the remote party to hand out the token and, thus, re-issuing a request to boot into the REE. The encrypted rootfs is not attested during early boot (Sec. 4.1). Thus, in the current implementation, downgrades of the encrypted rootfs are possible. Still, one can rely on existing features like dm-verity to attest the rootfs after one has booted the REE kernel, and, thus, prevent downgrades to the rootfs.

Memory-Based Attacks ($\mathfrak{T}4$). With a cold-boot attack, the adversary can only extract NW fragments from external RAM. This is because every trusted component runs on on-chip SW RAM. Still, the attacker can try to mount cold-boot attacks on the agent. The only secret the agent has indirect access to is the encrypted token E (Sec. 4.2). The encryption is based on K_T which is never exposed to the NW. DMA attacks are a further possible attack vector. However, recent ARMv8-A boards can configure DMA controllers to only allow NW DMA transfers.

Network-Based Attacks ($\mathfrak{T}5$). In Sec. 4.2, we analyze the key derivation protocol regarding the confidentiality of the token, the authenticity of the device and the trusted party, and the protocol's resistance to replay attacks. Besides the protocol, the hardware network stack might be vulnerable. Exploiting a vulnerability (or even replacing the peripheral), an adversary can try to send arbitrary SPI messages to the CPU. By comparing with a list of allowed SPI message patterns, we block unintended communication. We argue that this makes it difficult for an attacker to expand their privileges from the SPI peripheral to the device. But even if the attacker manages to do so, the agent is still isolated and has no access to secrets. The interface to the governor accepts only the response as a fixed-sized parameter, which makes vulnerabilities at this interface unlikely.

Logical Communication Attacks ($\mathfrak{T}6 - \mathfrak{T}7$). We argue that it is unlikely for an attacker to hijack the boot behavior via a logical communication interface that was not initialized by a driver before. We remove drivers that are not strictly required (e.g., USB) from the TCB. Our stripped-down version of U-Boot SPL has drivers for UART, I2C, and GPIO. TFA includes drivers for UART. The serial console prints

logs during boot. However, it only accepts input after the REE has booted. Thus, we assume that malicious boot-time input is not possible. Both I2C and GPIO are required to set up the DRAM on the Nitrogen8M. We carefully read the code and verify that no user-provided input is read. Without input, we deem a modification of the boot flow unlikely. Special attention has to be paid to hardware-based debugging interfaces like JTAG, allowing full access to the hardware including the TEE RAM. At the site of deployment, we require debugging interfaces to be permanently disabled. Contemporary hardware like the Nitrogen8M provides the means to do so.

6.3 Real World Attacks

Cold-boot attacks on keys in RAM are possible (Halderman et al., 2008; Gruhn and Müller, 2013). For example, File-Based Encryption (FBE) on Android was vulnerable (Groß et al., 2021). To mitigate this, one can store the keys in CPU registers (Müller et al., 2011; Simmons, 2011) or use on-chip RAM as we do.

According to CVEDetails, there are currently 2,839 CVEs in the Linux kernel, 34 in U-Boot, and 2 in TFA. Large-scale attacks like Mirai (Antonakakis et al., 2017) and Hajime (Herwig et al., 2019) show that outdated software is a threat to IoT and edge deployments. Moreover, IoT ransomware has gained momentum (Xu et al., 2019), showing that attackers increasingly target data on IoT and edge devices. Unfortunately, physical attacks (e.g., device theft) on large-scale existing edge deployments like one of communication network providers are hardly publicly known due to their critical nature. Therefore, we compare it with jailbreaking smartphones. Over the last decade, several critical vulnerabilities enabled jailbreaks. Without claiming completeness, a few current are CVE-2021-1782, CVE-2021-30883, CVE-2020-0069, CVE-2019-9467, CVE-2019-9436, and CVE-2019-8900 (axi0mX, 2019). The latter two demonstrate the extensive capabilities of vulnerabilities in early-boot software in combination with physical access. Security bugs in `fastboot` (a flashing tool for Android) allowed data exfiltration (Hay, 2017) and novel techniques to find vulnerabilities in bootloaders resulted in several new CVEs (Redini et al., 2017). While we do not remove vulnerabilities, our system helps to ensure the confidentiality of the data after theft and even in presence of said vulnerabilities. As soon as a software state is marked as outdated by the trusted party (e.g., because a vulnerability has become public), the REE cannot boot any longer. Even data exfiltration by exploiting bootROM vulnerabilities like Checkm8 (axi0mX, 2019) can be mitigated as soon a device theft has been detected and as long as the cryptography holds. This is because the trusted party can refuse to hand out the token from that point in time onward. Thus, the key material is just not physically available on the device. We anticipate more data-oriented attacks (incl. physical ones) in the future and argue that our system is a meaningful step toward a more secure edge infrastructure.

Table 2: Evaluation of the Crypto Performance.

Experiment	Enc. [MiB/s]	Dec. [MiB/s]
Linux AES	221.46 ± 3.52	222.12 ± 3.71
Key Shield	218.67 ± 4.26	219.11 ± 4.40
Serpent	31.86 ± 0.07	34.49 ± 0.09
Twofish	48.93 ± 0.20	50.24 ± 0.19
Adiantum	79.23 ± 0.38	79.24 ± 0.36

7 EVALUATION

We evaluate the performance costs of the components.
Cryptography Performance. We compare the key shield to the default Linux AES-XTS-512 implementation. Tab. 2 shows the results. The overhead of the key shield consists of trapping to TFA before a cryptography operation and exchanging the key with an alias key (Sec. 5). The AES-XTS-512 implementation is ported from Linux. We execute `cryptsetup benchmark` (version 2.3.4) on the Nitrogen8M 32 times and calculate the mean and the standard deviation. The benchmark performs the encryption and decryption on 4096-byte blocks directly on RAM, with no actual storage peripherals involved. The first block shows that the key shield slightly decreases the performance by 1.26 % (encryption) and 1.36 % (decryption).

The AES-XTS-512 algorithm relies on ARMv8-A hardware extensions for improved performance, which is visible when looking at pure software implementations of Serpent, Twofish, and Adiantum in the Linux kernel (second block). In light of this, an overhead of below 2% seems justifiable.
Boot Time. We implement the agent on a Raspberry Pi 4 (Sec. 5). In addition to that, we port the emergency switch and the governor as well. We measure the boot time to evaluate the costs of retrieving the token from the trusted party. We use a system timer (`CNTPCT_EL0`) to measure the time between the start of the TFA to the execution of the init process of the initfs (Sec. 4). Our baseline is a system that does neither retrieve a token nor derive an FDE key. We measure the retrieval of the token from the same LAN and on the internet. We boot the device 50 times each and

calculate the mean and the standard deviation. In the same LAN, we determine an overhead of 1.238s on average. Receiving the token from the internet leads to 1.253s of average additional boot time. The standard deviation is $\approx 1.5ms$ in both cases. Note that the boot time evaluation does not account for the confinement module. This is because the Raspberry Pi does neither have a TZASC nor a TZPC. The confinement module mostly consists of writes to memory-mapped registers. We assume that it does not significantly affect our network-focused evaluation. To verify, we measure the confinement module's runtime on the Nitrogen8M using the same system timer. Setting up and tearing down the confinement takes 0.015s each on average (16 samples), with an insignificant standard deviation. We conclude that $\approx 1.3s$ additional boot time in total is a realistic approximation. Relying on the redundancy of the edge computing paradigm, we deem this boot time increase acceptable.

8 RELATED WORK

There are several streams of related work.

Full Disk Encryption. FDE keys in external RAM are typically susceptible to cold-boot attacks (Halderman et al., 2008; Götzfried and Müller, 2013; Gruhn and Müller, 2013). Müller et al. and Simmons implement FDE without relying on external RAM as a mitigation (Müller et al., 2011; Simmons, 2011). In contrast, we use on-chip memory and a TEE to protect key material. Commercial OSs like Linux (dm-crypt), Microsoft Windows (Bitlocker), and Apple macOS (FileVault) support FDE.

File-Based Encryption. Recently, FBE has emerged as an alternative to FDE (Galindo et al., 2021). Being able to execute certain functions without a password (e.g., emergency calls on smartphones) is one of the main advantages. Groß et al. show that widely-available implementations leak metadata and key material (Groß et al., 2019; Groß et al., 2021). For this reason, we choose FDE in this paper.

Modifications to the Boot Process. A stream of work focusing on system resiliency and secure updates is most closely related. Xu et al. design a system that allows remotely recovering from a system compromise (Xu et al., 2019). They rely on a software network stack and storage devices that support hardware write protection. While Xu et al. assume a remote adversary, we deal with physical device theft. Huber et al. design a recovery system for low-end IoT devices (Huber et al., 2020). Quite similar to us, they use a hardware network stack. They propose software handlers in the SW as a proxy for critical peripherals. We disallow access to critical peripherals during early boot completely. Suzaki et al. propose a Linux-based network bootloader to download file system images (Suzaki et al., 2020). In contrast, we design our system to keep the attack surface small. We use a hardware network stack and design a lightweight protocol based on UDP. We focus on boot-time attacks and bind FDE to the most up-to-date system software.

TrustZone-Based Isolation Techniques. The research community proposed several new ways to utilize TEEs for software isolation and protection (Azab et al., 2014; Guan et al., 2017; Brasser et al., 2019). While those works focus on runtime isolation of software components, we adjust the boot process to achieve remote attestation and FDE key derivation. Similar to our work, Guan et al. use the TZASC to restrict access to RAM regions *temporarily* (Guan et al., 2017). We extend this concept to peripherals.

Remote Attestation. Remote attestation is a research field on its own (Kuang et al., 2022). We do not aim to develop new attestation techniques. Instead, we rely on existing techniques based on a hardware RoT.

9 LIMITATIONS

Our approach is limited to a boot-time attacker that steals the device. We assume that the adversary refrains from mounting runtime attacks at the site of deployment. If this does not hold, further attacks are possible. For example, she can try to take over the system while the REE runs. Intrusion detection systems are tailored for that scenario. However, those are research fields on their own and are considered orthogonal to our approach (Röckl et al., 2021; Liao et al., 2013). As already dealt with in Sec. 4.4, an early-boot network stack might hinder availability. However, being close to the network edge, the device can be protected by a wide range of existing (D)DoS defensive techniques (Cloudflare Inc., 2022).

10 CONCLUSION

We design a novel system architecture that protects data on stolen edge nodes. If a theft has been detected, the data on the device remains confidential as long as the cryptography is secure. We combine remote attestation and FDE in a way that a trusted party can first assert that the system software on the device is up-to-date and, second, actively decide whether the device is allowed to boot. This way, deprecated and potentially vulnerable early-boot system software cannot decrypt the disk - a feature that exist-

ing and widely distributed FDE implementations like Microsoft's BitLocker and Linux's dm-crypt lack.

We implement a prototype of our concept on two recent ARMv8-A devices. Our evaluation shows that the overhead is almost neglectable in practice, while we only add a low amount of unverified code to the TCB. Therefore, we believe that our architecture is a meaningful step towards future edge infrastructure.

ACKNOWLEDGEMENTS

This research was supported by the German Federal Ministry of Education and Research (BMBF) as part of the CELTIC-NEXT project AI-NET-ANTILLAS ("Automated Network Telecom Infrastructure with inteLLigent Autonomous Systems", Förderkennzeichen "16KIS1314").

REFERENCES

Android (2022). Android Keystore System. https://developer.android.com/training/articles/keystore. Accessed 2022-02-06.

Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J. A., Invernizzi, L., Kallitsis, M., Kumar, D., Lever, C., Ma, Z., Mason, J., Menscher, D., Seaman, C., Sullivan, N., Thomas, K., and Zhou, Y. (2017). Understanding the mirai botnet. In Kirda, E. and Ristenpart, T., editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1093–1110. USENIX Association.

ARM Limited. SMC Calling Convention System Software on ARM Platforms. https://developer.arm.com/documentation/den0028/a/. Accessed 2022-02-02.

ARM Limited. Trusted Board Boot Requirements Client (TBBR-CLIENT) Armv8-A. https://developer.arm.com/documentation/den0006/latest. Accessed 2022-02-03.

axi0mX (2019). Open-source jailbreaking tool for many iOS devices. https://github.com/axi0mX/ipwndfu. Accessed 2022-03-01.

Azab, A. M., Ning, P., Shah, J., Chen, Q., Bhutkar, R., Ganesh, G., Ma, J., and Shen, W. (2014). Hypervision across worlds: Real-time kernel protection from the ARM trustzone secure world. In Ahn, G., Yung, M., and Li, N., editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 90–102. ACM.

Brasser, F., Gens, D., Jauernig, P., Sadeghi, A., and Stapf, E. (2019). SANCTUARY: arming trustzone with user-space enclaves. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society.

Busch, M., Schlenk, R., and Heckel, H. (2019). Teemo: Trusted peripheral monitoring for optical networks and beyond. In *Proceedings of the 4th Workshop on System Software for Trusted Execution - SysTEX '19*. ACM Press.

Cao, K., Liu, Y., Meng, G., and Sun, Q. (2020). An overview on edge computing research. *IEEE Access*, 8:85714–85728.

Cloudflare Inc. (2022). Cloudflare DDoS Protection. https://www.cloudflare.com/en-au/ddos-de/. Accessed 2022-04-25.

Forescout Research Labs (2020). How TCP/IP Stacks Breed Critical Vulnerabilities in IoT, OT and IT Devices. https://www.forescout.com/company/resources/amnesia33-how-tcp-ip-stacks-breed-critical-vulnerabilities-in-iot-ot-and-it-devices/. Accessed 2022-02-05.

Galindo, D., Liu, J., Stone, C. M., and Ordean, M. (2021). SoK: Untangling File-based Encryption on Mobile Devices. arXiv.

Götzfried, J. and Müller, T. (2013). ARMORED: Cpu-bound encryption for android-driven ARM devices. In *2013 International Conference on Availability, Reliability and Security, ARES 2013, Regensburg, Germany, September 2-6, 2013*, pages 161–168. IEEE Computer Society.

Groß, T., Busch, M., and Müller, T. (2021). One key to rule them all: Recovering the master key from ram to break android's file-based encryption. *Forensic Science International: Digital Investigation*, 36:301113. DFRWS 2021 EU - Selected Papers and Extended Abstracts of the Eighth Annual DFRWS Europe Conference.

Groß, T., Ahmadova, M., and Müller, T. (2019). Analyzing android's file-based encryption: Information leakage through unencrypted metadata. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ARES '19, New York, NY, USA. Association for Computing Machinery.

Gruhn, M. and Müller, T. (2013). On the practicability of cold boot attacks. In *2013 International Conference on Availability, Reliability and Security, ARES 2013, Regensburg, Germany, September 2-6, 2013*, pages 390–397. IEEE Computer Society.

Guan, L., Liu, P., Xing, X., Ge, X., Zhang, S., Yu, M., and Jaeger, T. (2017). Trustshadow: Secure execution of unmodified applications with ARM trustzone. In Choudhury, T., Ko, S. Y., Campbell, A., and Ganesan, D., editors, *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'17, Niagara Falls, NY, USA, June 19-23, 2017*, pages 488–501. ACM.

Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., and Felten, E. W. (2008). Lest we remember: Cold boot attacks on encryption keys. In van Oorschot, P. C., editor, *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008,*

San Jose, CA, USA, pages 45–60. USENIX Association.

Harchol, Y., Mushtaq, A., Fang, V., McCauley, J., Panda, A., and Shenker, S. (2020). Making edge-computing resilient. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 253–266, New York, NY, USA. Association for Computing Machinery.

Hay, R. (2017). fastboot oem vuln: Android bootloader vulnerabilities in vendor customizations. In Enck, W. and Mulliner, C., editors, *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017*. USENIX Association.

Herwig, S., Harvey, K., Hughey, G., Roberts, R., and Levin, D. (2019). Measurement and analysis of hajime, a peer-to-peer iot botnet. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society.

Huber, M., Hristozov, S., Ott, S., Sarafov, V., and Peinado, M. (2020). The lazarus effect: Healing compromised devices in the internet of small things. In Sun, H., Shieh, S., Gu, G., and Ateniese, G., editors, *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020*, pages 6–19. ACM.

Kol, M. and Oberman, S. (2020). Ripple20. https://www.jsof-tech.com/wp-content/uploads/2020/06/JSOF_Ripple20_Technical_Whitepaper_June20.pdf. Accessed 2022-03-01.

Kuang, B., Fu, A., Susilo, W., Yu, S., and Gao, Y. (2022). A survey of remote attestation in internet of things: Attacks, countermeasures, and prospects. *Comput. Secur.*, 112:102498.

Liao, H.-J., Richard Lin, C.-H., Lin, Y.-C., and Tung, K.-Y. (2013). Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24.

Linaro Limited (2022). Open Portable Trusted Execution Environment. https://www.op-tee.org/. Accessed 2022-08-17.

Müller, T., Freiling, F. C., and Dewald, A. (2011). TRESOR runs encryption securely outside RAM. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association.

NXP Semiconductors (2022). Security Reference Manual for i.MX 8M Dual/8M QuadLite/8M Quad. https://www.nxp.com/webapp/Download?colCode=IMX8MDQLQSRM\&appType=moderatedWithoutFAE.

Pinto, S. and Santos, N. (2019). Demystifying arm trustzone: A comprehensive survey. *ACM Comput. Surv.*, 51(6):130:1–130:36.

Redini, N., Machiry, A., Das, D., Fratantonio, Y., Bianchi, A., Gustafson, E., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. (2017). Bootstomp: On the security of bootloaders in mobile devices. In Kirda, E. and Ristenpart, T., editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 781–798. USENIX Association.

Röckl, J., Protsenko, M., Huber, M., Müller, T., and Freiling, F. C. (2021). Advanced system resiliency based on virtualization techniques for iot devices. In *ACSAC '21: Annual Computer Security Applications Conference, Virtual Event, USA, December 6 - 10, 2021*, pages 455–467. ACM.

Selmke, B., Heyszl, J., and Sigl, G. (2016). Attack on a DFA protected AES by simultaneous laser fault injections. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, Santa Barbara, CA, USA, August 16, 2016*, pages 36–46. IEEE Computer Society.

Simmons, P. (2011). Security through amnesia: a software-based solution to the cold boot attack on disk encryption. In Zakon, R. H., McDermott, J. P., and Locasto, M. E., editors, *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011*, pages 73–82. ACM.

Suzaki, K., Tsukamoto, A., Green, A., and Mannan, M. (2020). Reboot-oriented iot: Life cycle management in trusted execution environment for disposable iot devices. In *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020*, pages 428–441. ACM.

Symantec (2016). Internet Security Threat Report. https://docs.broadcom.com/doc/istr-16-april-volume-21-en. Accessed 2020-05-11.

Syverson, P. F. (1994). A taxonomy of replay attacks. In *Seventh IEEE Computer Security Foundations Workshop - CSFW'94, Franconia, New Hampshire, USA, June 14-16, 1994, Proceedings*, pages 187–191. IEEE Computer Society.

Tim Hummel (2017). FPGA eMMC Reader/Sniffer. https://timhummel.com/portfolio/fpga_emmc_reader_sniffer/. Accessed 2022-08-12.

Western Digital (2017). e.MMC Security Methods. https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/collateral/white-paper/white-paper-emmc-security.pdf. Accessed 2022-08-12.

Xu, M., Huber, M., Sun, Z., England, P., Peinado, M., Lee, S., Marochko, A., Mattoon, D., Spiger, R., and Thom, S. (2019). Dominance as a new trusted computing primitive for the internet of things. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1415–1430. IEEE.

Yu, W., Liang, F., He, X., Hatcher, W. G., Lu, C., Lin, J., and Yang, X. (2018). A survey on the edge computing for the internet of things. *IEEE Access*, 6:6900–6919.

Zinzindohoué, J. K., Bhargavan, K., Protzenko, J., and Beurdouche, B. (2017). Hacl*: A verified modern cryptographic library. In Thuraisingham, B., Evans, D., Malkin, T., and Xu, D., editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1789–1806. ACM.