Reboot-Oriented IoT: Life Cycle Management in Trusted Execution Environment for Disposable IoT devices

Kuniyasu Suzaki
National Institute of
Advanced Industrial Science and Technology
k.suzaki@aist.go.jp

Akira Tsukamoto
National Institute of
Advanced Industrial Science and Technology
akira.tsukamoto@aist.go.jp

Andy Green
Warmcat
andy@warmcat.com

Mohammad Mannan
Concordia University
m.mannan@concordia.ca

ABSTRACT

Many IoT devices are geographically distributed without human administrators, which are maintained by a remote server to enforce security updates, ideally through machine-to-machine (M2M) management. However, malware often terminates the remote control mechanism immediately after compromise and hijacks the device completely. The compromised device has no way to recover and becomes part of a botnet. Even if the IoT device remains uncompromised, it is required to update due to recall or other reasons. In addition, the device is desired to be automatically disposable after the expiration of its service, software, or device hardware to prevent being cyber debris.

We present Reboot-Oriented IoT (`RO-IoT`), which updates the total OS image autonomously to recover from compromise (rootkit or otherwise), and manages the life cycle of the device using Trusted Execution Environment (TEE) and PKI-based certificates (i.e., CA, server, and client certificates which are linked to device, software, and service). RO-IoT is composed of three TEE-protected components: the secure network bootloader, periodic memory forensics, and life cycle management. The secure network bootloader downloads and verifies the OS image by the TEE. The periodic memory forensics causes a hardware system-reset (i.e., reboot) after detecting any un-registered binary or a time-out, which depends on a TEE-protected watchdog timer. The life cycle management checks the expiration of PKI-based certificates for the device, software, and service, and deactivates the device if necessary. These features complement each other, and all binaries and certificates are encrypted or protected by TEE. We implemented a prototype of RO-IoT on an ARM Hikey board with the open source trusted OS OP-TEE. The design and implementation take account of availability (over 99.9%) and scalability (less than 100MB traffic for a full OS update, and estimated at a cent per device), making the current prototype specifically suitable for the *AI-Edge* (Artificial Intelligence on the Edge) IoT devices.

ACSAC 2020, December 7–11, 2020, Austin, USA

ACM ISBN 978-1-4503-8858-0/20/12…$15.00
https://doi.org/10.1145/3427228.3427293

CCS CONCEPTS

• **Security and privacy** → **Operating systems security**; • **Computer systems organization** → *Embedded systems*; • → **Security in hardware**.

KEYWORDS

Reboot-Oriented IoT, Trusted Execution Environment (TEE), Public Key Infrastructure (PKI), Life Cycle Management

ACM Reference Format:
Kuniyasu Suzaki, Akira Tsukamoto, Andy Green, and Mohammad Mannan. 2020. Reboot-Oriented IoT: Life Cycle Management in Trusted Execution Environment for Disposable IoT devices. In *Annual Computer Security Applications Conference (ACSAC 2020), December 7–11, 2020, Austin, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3427228.3427293

1 INTRODUCTION

Many Internet of Things (IoTs) are geologically distributed, and used for AI-Edge to enable a wide range of applications such as crowd tracking, traffic monitoring, and weather forecasting smart cities [73, 95] and smart farming [104]. The AI-Edge IoT Devices are managed by Edge-Computing [24, 87, 92, 110] or Fog-Computing [69, 83], which decouples computing resources into a cloud and IoT devices, and maintains many IoT devices by a remote server automatically through a machine-to-machine (M2M) management system. These IoT devices collect and process large amounts of data to reduce the network traffic and computation on the back-end servers. These devices are maintained via Over-The-Air (OTA) updates [23, 56, 77] to reduce end-user responsibility.

Edge- and Fog-Computing consider the power consumption needs of IoT devices. Past work indicated that solar power can support long-term functioning of IoT devices (e.g., Raspberry Pi [5, 94, 98]). Disposability of physical IoT devices is also an essential factor and has been discussed through biologically self-destruction [37, 62] and physical collection policies defined by ITU E-Waste [48].

Most IoTs run a few domain-specific applications, mostly on Linux, due to the availability of many development tools on Linux. Although the Linux kernel itself is well maintained, and bugs are often fixed quickly [1, 51, 99, 105] (e.g., for Meltdown [66, 67] and Spectre [58, 59]), most Linux IoT devices are not updated accordingly, and become an easy target of attacks. Notable example attacks include: Mirai [9, 60], Linux.Darlloz [19, 108], BASHLITE [70], and Hajime [34]. Some attacks terminate the remote control mechanism

immediately after compromise, and hijack the devices to use them as part of a botnet. In addition, malware is also often designed to be persistent across device-reboot and hide its presence as a rootkit. Such compromised IoT devices are difficult-to-detect/restore.

IoTs are also subject to a product recall or urgent security patch when a severe vulnerability is found; e.g., the recall incident of child-tracking smartwatches [18] and web cameras [88]. IoT device providers also ideally want to deactivate the device after the expiration of the maintenance period. When IoT devices are no longer supported by their providers, they may eventually become *rogue IoT* devices, or simply, *cyber debris*.

To alleviate similar issues, several academic solutions have been proposed as system-reset architectures, including: Crash Only Software [25], MicroReboot [26], Let It Crash [11], collective[29], Misery digraphs [7], ReSecure [3], Restart-Based Protection [2], YOLO [13], CIDER [106], and TPM2.0's Authenticated Countdown Timer [97]. However, these are mostly reactive solutions and cannot adequately handle undetected or unknown malware. IoT device compromise and management issues are also being actively considered in industry/policy forums. For example, the IEEE Internet Technology Policy Community [45] suggested that device manufacturers should provide a "formal plan for users to sanitize and dispose of obsolete IoT devices." The MITRE Challenge IoT competition [74] also included the detection of rogue IoT devices. However, the IoT life cycle depends on long supply chains that include many stakeholders, which makes this problem difficult to solve, cf. Wang et al. [103].

We propose *Reboot-Oriented IoT (RO-IoT)* to address these problems by extending existing system-reset architectures [7, 11, 25, 26, 29, 97, 106]. RO-IoT consists of network boot, live memory forensics, and life cycle management—all of which are protected by the Trusted Execution Environment (TEE) [28, 76, 85, 90] to protect critical RO-IoT components from the untrusted OS. We utilize rebooting and reinstalling the OS as an essential mechanism to recover from a compromised system. In addition, the life cycles of device, software, and service are linked to Public Key Infrastructure (PKI) based TLS certificates (i.e., certificates for CA, server, and client). It means that the IoT device can boot only if the TLS connection is established for a secure network boot. Other difference between RO-IoT and past system-reset architecture is that the past projects are based on reactive protection, but RO-IoT is based on proactive protection. Therefore, RO-IoT causes occasional rebooting and recovers from an undetected or unknown compromise. In addition, the occasional rebooting also enables emergency security patching and product recalls. When the rebooting process detects the expiration of PKI-based certificates, RO-IoT makes the device *safely* disposable—the TEE and bootloader images on the storage are encrypted by hardware-protected keys.

In terms of deployability, RO-IoT poses two major challenges: suspension caused by occasional reboots, and power consumption. The suspension can be accommodated by our target AI-Edge applications, although more careful evaluation is needed for mission-critical and real-time applications. The power consumption needs in IoT devices have been explored by Edge-Computing and Fog-Computing. Our current prototype is based on a HiKey board, which has a similar power profile as a Raspberry Pi that can be permanently run using solar power [5, 94, 98]. The backgrounds of these challenges are discussed more in Section 3.

Contributions and Challenges:

(1) **Reboot-oriented IoT:** RO-IoT manages the IoT security and life cycle. It is tailored for IoT characteristics such as a limited number of applications, small OS image, and quick booting. The architecture aims to recover from a compromise (known or *unknown*) by autonomous rebooting, and guarantees safe termination of the device after the expiration of service, software, and device supports, which are linked by PKI-based certificates.

(2) **Secure network bootloader protected by TEE:** The secure network bootloader establishes a TLS connection from the protected TEE (i.e., ARM TrustZone secure world) and downloads an OS image. The certificate of the OS image is verified in the TEE, and the OS is booted in the normal world. As the total OS is renewed, we enable full recovery from a compromised OS. The network boot is autonomous and periodic. The bootloader uses a cached image when the OS is not required to be renewed, and thus saves the network traffic and boot time.

(3) **TEE-protected live memory forensics:** Periodic memory forensics of the normal OS is performed in the TEE and involves application whitelisting. The periodic activation of memory forensics utilizes a *watchdog* timer, which is also protected by TEE. RO-IoT sets a short period (e.g., 30 seconds) for the watchdog timer and enforces the *time-extend-request* periodically, which is used as a trigger for periodic memory forensics. When the request counter reaches the threshold, RO-IoT enforces the occasional system-reset to renew the OS, if necessary; it also serves as a trigger to check the life cycle. (Note: "*occasional*" system-reset requires a much longer time period than "*periodical*" memory forensics.)

(4) **IoT life cycle management protected by TEE and PKI-based certificates:** At the provisioning of the supply chain, RO-IoT clearly assigns responsibility for each stakeholder: the IoT device, software, and service providers. Each life cycle element is linked to a certificate of CA, server, or client used in TLS. The TEE manages each life cycle element, and makes the device inoperable when one of them is expired. After the expiration, the IoT device becomes physically disposable in a safe manner because software and data in the storage are encrypted with a key protected physically (i.e., SoC Key).

(5) **Implementation of RO-IoT:** The implementation of RO-IoT utilizes a small Linux image as the bootloader, which collaborates with *OP-TEE* [64, 79], a trusted OS for ARM TrustZone [28, 85]. Although the secure boot loader requires functionalities such as the management of TEE, secure networking, secure storage, and self-update, they are easily enabled by the small Linux. Linux is also well maintained [1, 51, 99, 105], offers a kernel self protection mechanism [32, 33, 54], and can be small in size [57, 61, 65, 80]. From the view of Trusted Computing Base (TCB), these features are perhaps more important than the size of code. Availability and scalability are also important features, and our implementation takes them into account. The security features and performance were evaluated on a LeMaker HiKey board with ARM64 HiSilicon Kirin 620 chip with 2GB memory. Our experiments showed the secure network boot took about

21 seconds on 14MB Minimal Linux. Even if the network rebooting occurs after 10,000 memory forensics (42 hours), the availability is 99.986%.

2 ATTACK TYPES AND THREAT MODEL

This section gives a brief description of attack types we consider on M2M IoT device and our threat model.

2.1 Attack Types

We assume that attackers can easily find vulnerable M2M IoT devices, including rogue IoT (i.e., cyber debris), using dedicated search engines designed for discovering Internet-connected IoT devices (e.g., Shodan [93], Censys [27], and ARE [10, 36]). Many IoT malware instances (e.g., Mirai variants [9, 19, 34, 60, 70, 108]) exploit weak security configurations, or known vulnerabilities. After the exploit, they hijack the remote control mechanism (e.g., by changing the SSH password) and run a downloaded tool to launch DDoS attacks.

Some malware may also persist on the device after rebooting, by infecting the file system or bootloader. Infections can be subtle and very difficult to detect (e.g., Subvirt [55] and Blue Pill [89]). Complete recovery from such persistent infection is very challenging. An effective way is full OS reinstallation from scratch—for PC environments, cf. [47, 100, 101].[1] Although the overhead to reinstall a full OS must be taken into account, M2M IoT can leverage the domain-specific features, such as running a few applications, to bring the overhead to an acceptable level.

Runtime security is also important to (drastically) reduce the period of a compromise. However, achieving runtime security is a long-standing challenge. IoT can also utilize the domain-specific features, such as application whitelisting, which is effective for small systems (see e.g., [19, 81, 91]).

For non-fixable critical vulnerabilities, the suppliers might issue a product recall. However, 100% recall is difficult, and some of these devices may even be forgotten by the end-users, and eventually become cyber debris. We argue that IoT devices should be terminated at the expiration of their life cycle for security purposes, especially for pervasive/ubiquitous IoT devices that are not actively managed by users/admins.

Although modern TEE-enabled CPUs can enhance system security, flaws in TEEs are also not too uncommon. Many such issues are implementation bugs, but some are system intrinsic. For example, the boomerang attack [68] abuses a pointer operation in the ARM TrustZone's secure world. On the other hand, replay attacks [30, 71] abuse the creation procedure of TEE-application on the untrusted OS. RO-IoT must be carefully implemented to avoid these known vulnerabilities.

2.2 Threat Model and Security Assumptions

Security goals and assumptions for RO-IoT include the following:

(1) RO-IoT's main security goal is to prevent abuse of an M2M IoT device (e.g., DDoS caused by Mirai) and to ensure that the M2M IoT device is used only for its intended purposes.
(2) RO-IoT requires occasional system-reset and must suspend the service for a certain length (less than approximately 60 seconds). The application must be designed to compensate for the service suspension, for example, by multiplexing IoT devices. The primary target for the current RO-IoT prototype is AI-Edge, and the 60-second suspension is acceptable in these use cases (when booting happens occasionally/infrequently). We discuss mission critical IoT use cases in Section 7.4.
(3) The current RO-IoT prototype does not protect physical side channel attacks as ARM TrustZone lacks memory encryption (unlike Intel SGX).
(4) RO-IoT cannot protect attacks that use code from legitimate application and kernel binaries, e.g., Return Oriented Programs (ROPs). These vulnerabilities must be fixed by the occasional system-reset.
(5) Our current memory forensics implementation protects against library replacement attacks that use *LD_LIBRARY_PATH* and *LD_PRELOAD*, but does not detect dynamically changing code, i.e., self-modifying code, as enabled by e.g., dl_open(), plug-in, and JIT (Just In Time compiler). We discuss these issues more in Sections 6 and 7.1.
(6) RO-IoT assumes a private Certificate Authority (CA). Certificates for the CA, client, and server are used for the life cycle management for the device, software, service. Each certificate is set at each stakeholder, (i.e., device supplier, software vendor, and service provider).
(7) Current implementation on a LeMaker Hikey board has some hardware limitations (i.e., no SoC key and watchdog-timer access from the normal world, see Section 5.3.2).

[1] Note that a system backup image is useful for recovery from faults, but not from compromise as backup copies may also be infected.

3 RELATED WORK AND BACKGROUND

RO-IoT extends the system-reset architectures [7, 11, 25, 26, 29, 97, 106], and invokes network boot to replace the whole OS image occasionally (i.e., reinstalling).

To recover from failure, rebooting is a sound strategy for OS management. For example, Crash Only Software [25] and MicroReboot [26] show that crashing is the faster way to reboot the OS than a normal reboot procedure with recovery. Let It Crash [11] shows that crashing eliminates troublesome error handling.

From a security perspective, quick invalidation of a compromised system makes further abuse difficult. Additionally, a swift update reduces vulnerable time-window. For example, Misery digraphs [7] show that VM instances in a web service provider like Amazon can be periodically reset and replaced with updated OS to prevent intrusion attacks. ReSecure [3], Restart-Based Protection [2], and YOLO [13] show the same effectiveness on small cyber physical systems that run real-time OSes.

Similar to RO-IoT, TPM 2.0's Authenticated Countdown Timer (ACT) [97] and CIDER [106] enable secure system-reset. ACT causes a system-reset from a TPM chip, which works as a protected watchdog timer, but it does not guarantee an OS update after system-reset. On the other hand, CIDER uses TEE for implementing the authenticated watchdog timer (AWDT) only. RO-IoT uses TEE to protect some critical components: boot loader, memory forensics, and life cycle management. These three components cooperate with each

other. In addition, CIDER relies on administrators to discover a compromise, but RO-IoT causes a system-reset proactively (i.e., occasionally) to protect from an undetected and hijacked compromise. The TEE of RO-IoT also protects PKI-based certificates and a private key, which are used for life cycle management of device, software, and service.

Some types of malware infect the file system or bootloader to persist after rebooting. To counter such malware, RO-IoT uses an isolated network bootloader and runs an OS on the memory only, which includes a file system. The core idea is similar to the *Collective* [29], a cache-based system management architecture, but RO-IoT does not require virtual machines. RO-IoT is operated through a secure network bootloader protected by TEE.

On the other hand, RO-IoT must address the suspension time caused by occasional rebooting. AI-Edge applications targeted by RO-IoT generally collect and process a large amount of video or sensor data for various use cases, including: smart cities [73, 95] and smart farming [104]. However, the cameras and sensors are blocked by obstacles occasionally, causing data loss/unavailability. Fortunately, many distributed devices report their results to the cloud and the statistical processing on the cloud solves this missing data issue. Apparently, unavailability caused by occasional rebooting in RO-IoT is acceptable for such AI-Edge applications, as long as many devices are not rebooted at the same time (discussed more in Section 7.3).

Power consumption of IoT device is also essential factors for IoT deployment. One reason to introduce Edge-Computing [24, 87, 92, 110] and Fog-Computing [69, 83] is the distribution computing/network resources to IoT devices, instead of fully-centralized processing (requiring significant bandwidth and CPU costs). Distributed IoT devices must run with low power because some AI-Edge applications may be used for a long period. Some deployments can assume AC power supply, while others must function as standalone (i.e., no direct power line). Fortunately, past studies [5, 94, 98] showed that current solar power and battery could permanently run an IoT device with Raspberry Pi (ARM Cortex-A CPU, Linux OS); the current RO-IoT prototype uses similar hardware (LeMaker Hikey board) with Linux.

In addition, IoT device becomes inexpensive and used for single use [75],which makes them physically disposable after use. This movement may cause significant environmental issues. The most desired solution is a biologically self-destructive device [62] (see also *Internet of Disposable Things* [37]); however, these devices currently cannot run Linux and AI-Edge applications. The reasonable solution is physical collection by legal regulation and policy enforcement [14]; see also the E-Waste Handbook [48] from the International Telecommunication Union (ITU). The deployment of RO-IoT must follow such regulations and policies.

Other technologies required by RO-IoT are secure network bootloader [6, 46, 84], live memory forensics [43, 52, 86, 96, 102, 107], PKI [31, 35, 44], and TEE (e.g., ARM TrustZone) [15, 16, 28, 40, 50, 78, 82, 85].

4 DESIGN

RO-IOT makes IoT devices to reboot the OS occasionally and renew the OS from the server, if necessary. During the device operation, the memory is scanned periodically, and the IoT device reboots the OS if an unregistered binary is found. At boot time, three TLS certificates (server, client and root) are examined, and RO-IoT renders the device inoperable when any of them is expired; i.e., the IoT device can be active only if the TLS connection is established at boot time. RO-IoT is composed of a secure network bootloader, live memory forensics, and life cycle management. These components are protected by TEE. See Figure 1 for a design overview of RO-IoT.

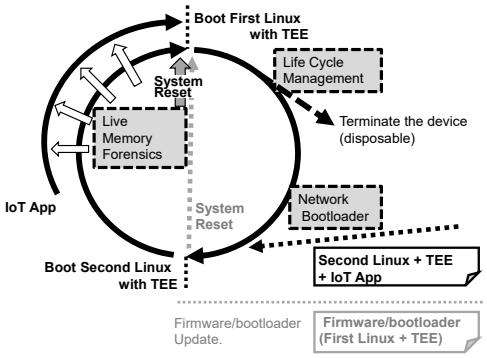

Figure 1: RO-IoT design overview. The dotted components are protected by TEE.

4.1 Two types of Linux on RO-IoT

RO-IoT uses two types of Linux images alternately. The first Linux works as a network bootloader and life cycle manager. The second Linux is downloaded from the server, and runs the IoT applications. The first Linux may be replaced by another boot loader (e.g., iPXE [46]). However, the bootloader for RO-IoT has some special requirements: 1) management of TEE (especially for OP-TEE); 2) secure networking to download a Linux image; 3) file system to save the Linux image as cache; and 4) self-update. These requirements are satisfied by most Linux distributions.

The two types of Linux have some different features. The first Linux allows the use of the storage, but the stored data is encrypted by the key included in the TEE. On the other hand, the second Linux kernel has no driver for storage and file system. In addition, the first Linux kernel includes *kexec* system call, which allows booting another Linux kernel. However, the second Linux kernel excludes it to stop anonymous kernel booting.

Along with the first and second Linux running on the normal world, a trusted application (TA i.e., TEE-application) runs on the secure world; *TA-Boot* and *TA-Forensics*, respectively. TA-Boot enables secure network booting and life cycle management. TA-Forensics allows live memory forensics of the second Linux.

The size of Linux kernel is another problem, but Linux has size reduction mechanisms (e.g., tinification [65, 80], Link Time Optimization (LTO) [57, 80], and undertaker [61]). RO-IoT utilizes the undertaker to make a small Linux kernel.

The bootloader included in the storage of IoT device (i.e., trusted boot firmware, secure monitor, OP-TEE, the first Linux, and PKI-based certificates) is assumed to be encrypted by the SoC key or a TEE protected key. Such a hardware-protected key makes the device disposable after the expiration of service.

4.2 Securing Network Boot

RO-IoT utilizes the secure network bootloader, which cooperates with live memory forensics and life cycle management.

4.2.1 Booting Mechanism with TEE. The booting mechanism cooperates with the TEE, and part of it is implemented in the *TA-Boot*. The URL of the download server for the second Linux and the client certificate are available to the TA-Boot application. The TLS connection is established between the download server and TA-Boot by mutual authentication with the TLS server and client certificates (the client private key is protected by TA-Boot). The download server identifies the IoT device by its unique client certificate and chooses the appropriate second Linux image. The Linux image, which includes the Linux kernel, root file system, and device tree, is also signed by a private signing key on the server and verified by a public verification key in TA-Boot.

4.2.2 Reusing Downloaded Linux Image. The downloaded Linux image is relatively small, but still repeated downloading affects the boot time. If a new Linux is not released, RO-IoT reuses the current image and omits the downloading. This mechanism requires the help of the first Linux because the TEE OS (i.e., OP-TEE) has no storage or file system. The downloaded Linux image is encrypted and stored in the file system of the first Linux; the encryption key is embedded in TA-Boot.

4.2.3 Updating Firmware/Bootloader. When a vulnerability in the firmware/bootloader is found, RO-IoT allows updating the firmware/bootloader itself (e.g., a set of trusted boot firmware, secure monitor, and OP-TEE) from the back-end server. Especially, the bootloader may need frequent updates as it contains a Linux kernel. The bootloader updates the firmware/bootloader image on the storage while it runs on memory only. The secure connection and image verification are performed by TA-Boot. All secret keys used by RO-IoT can be renewed except for the SoC key.

4.2.4 Scalability. The OS image for the IoT is small (currently under 100MB). We assume even if an emergency update is required by a lot of devices, the total traffic volume will remain manageable, considering the high-capacity of existing content delivery networks (CDNs). For example, if 10,000 devices are managed by a provider, the total network traffic is 1TB, which costs about 100 USD (i.e., 1 cent for a full OS update on a device) on Amazon AWS [8] and Google Cloud Platform [42].

4.2.5 Availability. The period of occasional system-reset and reboot time resemble to be Mean Time Between Failure (MTBF) and Mean Time To Repair (MTTR). The availability is estimated to be 99.93% when a system-reset is caused every day, and network rebooting time is 60 seconds. RO-IoT has a mechanism to short the reboot time by reusing the cached image when there is no update. It means that the availability can be higher.

4.3 Live Memory Forensics

RO-IoT offers live memory forensics, enabling application whitelisting in the second Linux. In general, memory introspection, especially virtual machine introspection (VMI) [17, 20, 38], is a technique to analyze live applications. However, memory introspection works as a general tool, and includes functions of debugging and monitoring. RO-IoT is designed to use memory forensics only in TEE to protect integrity of the forensics process.

4.3.1 Hash Database for Memory Forensics. At the setup phase, a hash database for each 4KB text page (i.e., code) of an ELF binary is created. The database includes hashes for the dynamic linking libraries to prevent library replacement attacks. Therefore, the analysis tool traces the dynamic linking libraries used by the ELF file. Unfortunately, the tool cannot trace the library opened dynamically (e.g., dl_open() or plug-in) because the library name depends on the binary context and cannot be determined by the linking information statically. As such, our current RO-IoT prototype does not allow applications which include dl_open() and related functions.

The hash database is stored in TA-Forensics which scans the memory of the second Linux periodically. All running processes are subjected to page hash verification. If a page in memory does not match a hash, e.g., unregistered or compromised ELF binary, the TEE issues a system-reset. The page verification procedure may be targeted in a boomerang attack [68] as it allows pointer access to the normal world. The implementation takes such attacks into consideration (see Section 5.2.2).

The hash database depends on the downloaded image, and it must be set in the TEE securely before the second Linux boots. This issue is addressed by OP-TEE, by making a TA to survive *kexec* reboot (see Section 5.1.3).

4.3.2 Enforcement of the Periodic TA Service. A TA is a passive service because OP-TEE does not offer sleep and wake-up mechanism. To support periodic TA-Forensics, it must accept a periodic request from the normal world. However, the activation mechanism may be stopped by an attacker to terminate the memory forensics process. This is addressed by using a hardware *watchdog timer*, which is typically used to recover from malfunctions. The watchdog timer sets a short time length to issue a system-reset frequently. If the watchdog timer is not extended, the device enforces a system-reset. The watchdog timer is protected by TEE in a similar manner as in CIDER [106]. The time setup mechanism is allocated by TA-Forensics (i.e., in the TEE) only, and the untrusted OS must activate TA-Forensics periodically (as otherwise the operation in the normal OS, as well as trusted OS, is rebooted). The period of time-extend-request is short (e.g., 15 seconds) to trigger the memory forensics frequently.

4.3.3 Enforcement of Occasional System-Reset. TA-Forensics has a counter and increments the number when a time-extend-request comes from the normal world; it enforces a system-reset when the counter reaches a threshold. When the threshold is 10,000, and periodic time-extend-request is issued every 15 seconds, the occasional system-reset happens in every 42 hours. This mechanism is different from CIDER, which has no threshold. The occasional system-reset enforces the expiration of the device life cycle.

4.4 Life Cycle Management

The life cycle management for M2M IoT devices is more critical than normal devices (e.g., home IoT), because they are geographically distributed without human administrators. The relation of supply chain stakeholders is complex, but the lack of coordination between them leads to serious security issues (e.g., the lack of security patch

coordination). Therefore, a major goal for RO-IoT is to enable life-cycle management for M2M IoT devices.

RO-IoT assumes that the supply chain of IoT consists of four stakeholders (device factory, device supplier, software vendor, and service provider), and the last three stakeholders are responsible for the corresponding life cycle (RO-IoT assumes the responsibility of the device factory is covered by the device supplier). The three life cycles are linked to the expiration of three certificates used in TLS connections (i.e., CA, server, and client). *TA-Boot* verifies the certificates at boot time and deactivates the device to prevent abuse if necessary.

4.4.1 Life Cycle of the IoT hardware device. The expiry time of the device is managed by the device supplier. This relates to the warranty of the device, and the device supplier may want to stop the operation of the device after the expiration of repair terms and conditions (e.g., the device supplier may not want to offer security patches after a certain time period).

RO-IoT assumes that a device supplier creates their own private Certificate Authority (CA) for an SoC product. The CA certificate is embedded in the TA-Boot, and the expiration causes the SoC device to be inoperable. The pre-established expiry of the device prevents it from becoming cyber debris. An expired device can only be revived by the supplier, by replacing the firmware that is assumed to be encrypted by the SoC key.

4.4.2 Life Cycle of Software. The expiration of software is defined by the software vendors. RO-IoT assumes that the life cycle of the software is linked to a client certificate for TLS.

RO-IoT offers a provisioning mechanism for software vendors, which enables them to embed a client certificate in the TEE. The provisioning mechanism also embeds other critical data, i.e., a packaging public key to verify the second Linux, and the download URL which is offered by a service provider.

4.4.3 Life Cycle of Service. We assume that the service provider is responsible for the distribution of OS images to M2M IoT devices. The OS image is downloaded by HTTPS, and the server has a server certificate to establish a TLS connection. The expiration of the server certificate is used to control the expiry time of the service. This mechanism also enables the service provider to handle emergency security incidents (e.g., product recall).

5 IMPLEMENTATION

RO-IoT is implemented on a LeMaker Hikey board (ARM Cortex-A53 SoC Kirin 620, 64-bit 8-core 1.2GHz, 2GB Memory, and 8GB eMMC storage) with Linux and OP-TEE [64, 79]. As the board has no wired NIC, we add a 100Mbps Logitec USB 2.0 Adapter.

5.1 Secure Network Bootloader

Figure 2 shows an overview of the reboot (kexec or watchdog timer system-reset) cycle of RO-IoT. (Note: the upper of secure boot is the boot procedure defined by ARM TrustZone. The usage of TA-Client and TEE-Supplicant follows the manner of OP-TEE [64, 79]. It involves two phases: secure booting and normal operation (live memory forensics). Each phase has its own Linux in the normal world—termed as the *first* and *second Linux*. Each phase also has its own TA in the secure world: *TA-Boot* for secure network boot and

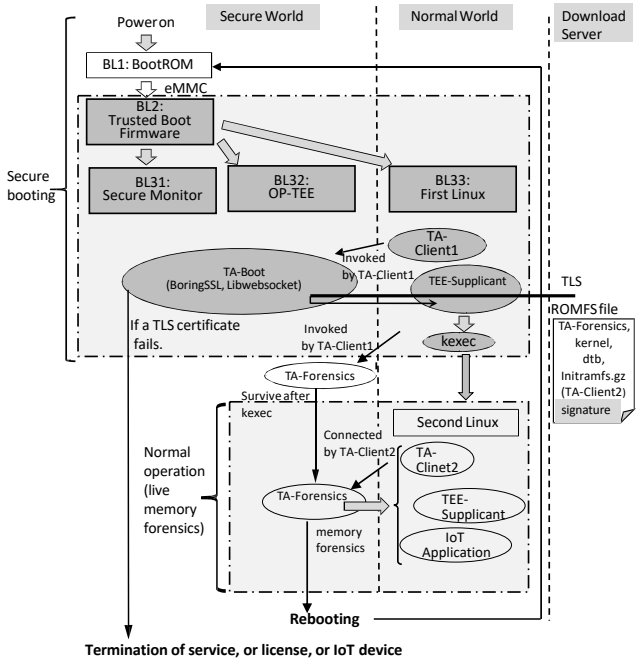

Figure 2: Overview of the reboot (kexec or system-reset) cycle of RO-IoT. Gray components are encrypted in the eMMC.

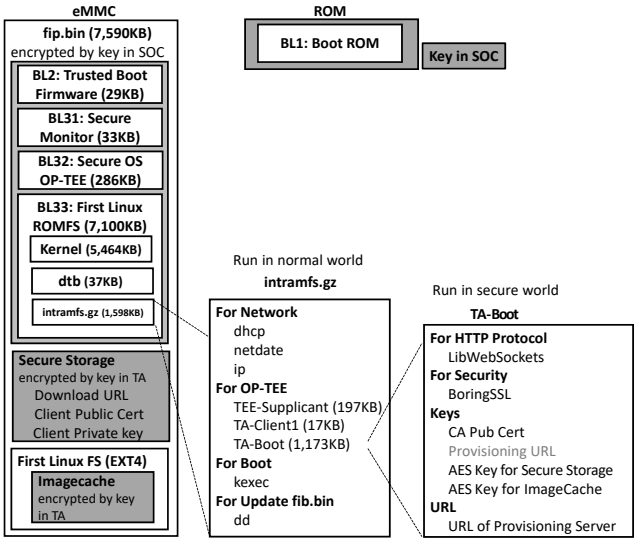

Figure 3: Software components embedded in the Hikey board. Gray components are encrypted by each key.

life cycle management, and *TA-Forensics* for live memory forensics, which are designed in section 4.

Figure 3 shows the software components embedded in the Hikey board. The boot ROM (Boot Loader1: BL1) is stored in ROM of the Hikey board, and the other components are stored in the eMMC. The eMMC has three secure areas: fib.bin, secure storage, and cache image; each area is encrypted by a different key: fib.bin is assumed

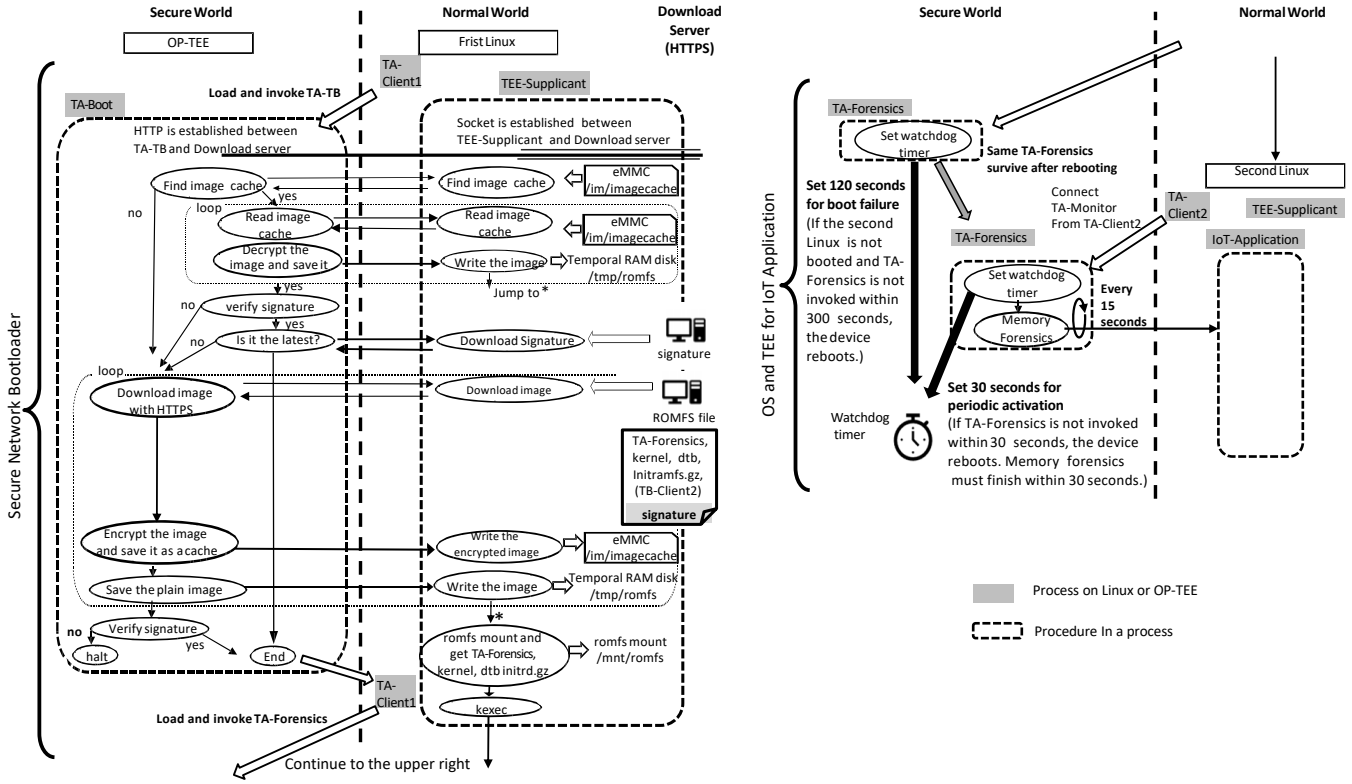

Figure 4: Detail steps for the secure network bootloader (left figure), and OS and TEE for the IoT application (right figure).

to be encrypted by the SoC key, and secure storage and cache image are encrypted by AES keys in TA-Boot.

5.1.1 Booting the First Linux. The boot ROM (BL1) first loads the "fip.bin" file to a fixed memory region. The fip.bin file includes all software used for the first Linux booting. The Trusted Boot Firmware (BL2) is invoked and loads Secure Monitor (BL31), OP-TEE (BL32), and the first Linux (BM33) image. The most part of the userland of the first Linux is occupied by networking tools (combined in a busybox), OP-TEE tools (`TEE-Supplicant`, `TA-Client1`, and `TA-Boot`), `kexec`, and dd commands.

The init script, the first program of first Linux, runs TA-Client1 and TEE-Supplicant. TA-Client1 loads TA-Boot on the secure world using the OP-TEE API *"TEE_IOC_OPEN_SESSION"*. TA-Boot accesses the secure storage with the help of TEE-Supplicant on the first Linux, which has drivers for the eMMC storage and EXT4 file system. TA-Boot gets the URL of the download server, client public certificate, and client private key from the secure storage, which are set at the provisioning phase (see Section 5.3).

5.1.2 HTTPS connection from TA-Boot. OP-TEE has no driver for the network interface, and thus TA-Boot cannot establish a network connection directly. TA-Boot uses TEE-Supplicant to relay a packet to the download server. The socket connection between the download server and TEE-Supplicant is established, i.e., Transport Layer Protocol (TLS). TA-Boot connects with TEE-Supplicant via OP-TEE and passes packets for TLS and HTTP to communicate with the download server. The packets are created by boringSSL [22] and libWebSocket [63] in TA boot. TA-Boot also has CA public and client certificates, and the TLS handshake (i.e., mutual TLS authentication) is performed.

5.1.3 Booting the Second Linux. Figure 4 depicts the booting process for the second Linux, performed in parallel while establishing the HTTPS connection. The contents for the second Linux are packed in a *ROMFS file*, which is a lightweight read-only loopback file system. The most important parts of booting are implemented in TA-Boot and TEE-Supplicant. The booting process consists of the following four phases.

(1) Check for a cached OS image TA-Boot checks the existence of an encrypted cache image (i.e., */im/imagecache* which is an encrypted ROMFS file) on eMMC's EXT4 file system, with the help of TEE-Supplicant. If the */im/imagecache* file does not exist, TA-Boot downloads a ROMFS file (see the next phase). If the */im/imagecache* file exists, TA-Boot decrypts it with imagecahce AES key embedded in TA-Boot (See Figure 3) and stores at */tmp/romfs* (i.e., ROMFS file) on the RAMFS of the first Linux with the help of TEE-Supplicant. The image cache is read speculatively, which can start before the network connection is established. The file's signature is also verified with the built-in packaging public key in TA-Boot. If the signature verification fails, TA-Boot downloads a ROMFS file. Even after successful verification, TA-Boot still downloads the packaging signature from the download server, to check if a new OS image is released. If the two signatures differ, TA-Boot downloads the

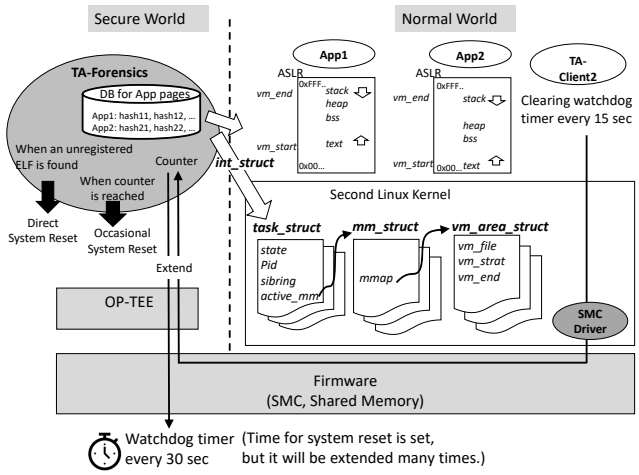

Figure 5: Overview of live memory forensics

new ROMFS file. If the signatures are the same, go to (5.1.3) Set up TA-Forensics.

(2) Downloading OS Image TA-Boot downloads the ROMFS file through HTTPS. However, the secure world has 16MB memory only and thus cannot hold the whole ROMFS file. Therefore, every 128KB data is downloaded by TA-Boot, and the data is stored in the file system of Linux via TEE-Supplicant. The data is saved in two files: */tmp/romfs* on RAMFS, and */im/imagecache* on EXT4 on eMMC, which is a cache image encrypted by the imagecache key.

(3) Set up TA-Forensics for the Second Linux The */tmp/romfs* (i.e., ROMFS file) is signed by the packaging private key on the server and is verified by the packaging public key in the secure storage (placed during setup). If the verification fails, the IoT device halts. If succeeded, the /tmp/romfs file is loopback-mounted, and the Linux kernel, initramfs.gz, dtb, and TA-Forensics are extracted. *TA-Forensics must be loaded and invoked by TA-Client1 on the first Linux* just before booting the second Linux because TA-Forensics verifies the applications on the second Linux. If TA-Forensics was invoked from the second Linux, the loading process might be compromised, or the database of whitelisted applications may be leaked. In addition, the launching of TA-Forensics should be hidden from the second Linux to avoid replay attacks [30, 71]. Fortunately, TA-Forensics can survive after the second Linux reboot of `kexec` on the normal world. *TA-Client2* on the second Linux can connect to TA-Forensics using the same UUID used by TA-Client1. TA-Forensics sets the watchdog timer with 120 seconds in the secure world. The set-time is longer than the memory forensics period because it includes the time for *kexec* booting. This mechanism also handles boot failures.

(4) Boot the Second Linux with kexec The kernel, dtb, and initramfs.gz are passed to *kexec* to boot the second Linux. At the booting, TA-Client2 connects to TA-Forensics with the UUID. TA-Forensics extends the watchdog timer with 30 seconds and starts memory forensics.

5.2 Live Memory Forensics

Figure 5 shows the outline of live memory forensics for the second Linux, which is performed by TA-Forensics.

5.2.1 Creating the database of ELF binaries. The hash database used by TA-Forensics is created by our "*elfcheck*" tool. It scans all ELF binaries in the second Linux. The elfcheck analyses the dynamically linked libraries with *ldd* and calculates a SHA256 hash (32 bytes) of each 4KB page (*.text* region). The database also includes the name of each ELF file to verify the name of a process.

At the kernel building time, RO-IoT detects the address of *init_struct* in the Linux kernel, which is the entry point of *task_struct*. All running processes on Linux are managed by the *task_struct* linked list. Each SHA256 hash of 4KB code page is compared with the database.

5.2.2 Memory forensics. When TA-Forensics runs, it looks at the running kernel's task tree in memory directly (i.e., from *init_struct* to *vm_area_struct* in Figure 5), without cooperation from the running kernel. For each running task, it tries to match the task by name with its database. Unknown ELF file names directly indicate a compromise. For known ELF file names, TA-Forensics hashes every executable page in the task's memory map and compares it with the hash database. Any unexpected executable pages indicate a compromise. Writable pages (i.e., bss and data) cannot be verified, since they may be changed arbitrarily.

When an ELF file is loaded into process memory, it will skip some sections present in the ELF image and may partially load (demand-load) others. Therefore, the footprint in memory of the executable or library is not wholly related to the layout in the original ELF file and may be dynamic as demand-paged content is added to or removed from memory. For that reason, validation must be done at page granularity with individual hashes. All pages marked with the "Executable" attribute are subject to verification. TA-Forensics issues a system-reset if it finds an unregistered memory hash.

To implement TA-Forensics, two new APIs, *TEE_Physmem()* and *TEE_Spinlock()*, are added to OP-TEE. TEE_Physmem() is used for translation from a virtual address to a physical address on the secure world. TEE_Spinlock() is used for the exclusive control of *task_struct*. During verification, TA-Forensics locks the *task_struct* of a process to prevent any concurrent update of process memory pages using kernel spinlock at the untrusted kernel side. It means that only the process being verified is blocked by TA-Forensics, and the rest of the processes can run along with TA-Forensics.

TEE_Physmem() and *TEE_Spinlock()* can be the attack surface of the boomerang attack [68]. However, they are limited to access the text region and *task_struct* only. Therefore, boomerang attacks are not effective against the current implementation. Furthermore, RO-IoT assumes that the IoT device has no sensitive information in the user space, and there is no threat to expose.

The memory forensics process does not cause a problem for ASLR (Address Space Layout Randomization) on the userland because TA-Forensics knows the entry point of *task_struct*. However, the kernel ASLR may cause a problem as it changes the entry point. Fortunately, current kexec on ARM does not support kernel ASLR. We leave this issue for future work.

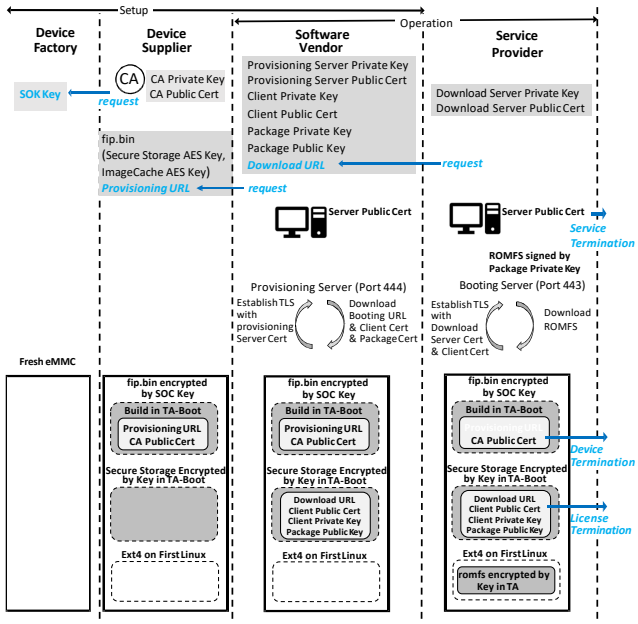

Figure 6: Setup and operation of the life cycle management

Table 1: Life cycle management keys and certificates based on PKI

Entity	Key type	Location	Function
CA	Private	Secret	Sign server and client certificates
	Public Cert	device supplier, HTTPS server, TA-Boot	Used by everyone to confirm certificates were created by the organization with CA private key
Server	Private	Secret	Used to confirm the server really has a certificate signed by CA
	Public Cert	HTTPS server	Send to clients to claim the server has a private key signed by CA
Client	Private	Secret	Used to confirm the client really has a certificate signed by CA
	Public Cert	Secure Storage	Send to the server to prove the client has a valid cert signed by CA

5.2.3 Periodic invocation of TA-Forensics. As mentioned in Section 4.3.2, RO-IoT uses a watchdog timer on the Hikey board (SP805 on Kirin 620) to enforce the periodic invocation of TA-Forensics. At every memory TA-Forensics, the timer is extended to prevent the hardware system-reset. An attacker is faced with two choices, i.e., let the memory forensics process run periodically, or change the system to stop the memory forensics process, and then face a system-reset.

The watchdog timer is set every 30 seconds by TA-Forensics (See lower-right Figure 4). The time-extend-request to extend the timer is operated from TA-Client2 every 15 seconds after the memory forensics process finishes. It means that the memory forensics process must finish within 15 seconds, and 15 seconds must be left for TA-Client2. Even if memory forensics fails for any reason, the watchdog timer issues a system-reset in 30 seconds.

5.3 Life Cycle Management

RO-IoT uses three TLS certificates for the CA, client, and server for life cycle management for IoT device, software, and service, respectively, as defined in Section 4.4. Each certificate is set up by the corresponding stakeholder and verified by *TA-Boot*. Table 1 shows the life cycle management keys and certificates, and Figure 6 shows an overview of the setup and operation of life cycle management.

5.3.1 Provisioning. RO-IoT provides a mechanism to provision the IoT device for setup and operation phases. At the setup phase, a *provisioning server* is operated by a software vendor and used to install the software on the IoT device. At the operation phase, a *download server* is operated by the service provider for a second Linux. Our current implementation assumes that the provisioning server uses port 444, and the downloading server uses port 443 for HTTPS to avoid confusion (see Figure 6).

5.3.2 Device Manufacturer. Usually, the specification of SoC is negotiated with the device supplier. Unfortunately, the HiKey board does not include an SoC key, although the specification of kirin 620 mentions that the setting depends on the eFUSE configuration. Therefore, our current implementation assumes that code and data on the eMMC are safe. In addition, the Hikey board does not limit the watchdog timer to the secure world only. Therefore, the watchdog timer can be set by an application in the normal world. In a real business situation, the SoC key must be installed, and the watchdog must be set from the secure-world only.

5.3.3 Device Supplier. The device supplier is responsible for the IoT device and its firmware. RO-IoT assumes a device supplier creates its own private Certificate Authority (CA) and offers a provisioning mechanism for the software vendor and service provider. The supplier includes the provisioning server's URL and CA certificate in the firmware (i.e., TA-Boot in fip.bin). The URL is set by the software vendor as it must prepare the provisioning server. The device supplier sets up the secure storage and EXT4 file system on the eMMC, accessed from the first Linux, and used by the software vendor and service provider, respectively. TA-Boot includes 128-bit AES keys for secure storage and imagecache. The two keys are used in TA-Boot only and do not need to be exposed to other stakeholders.

5.3.4 Software Vendor. The software vendor is responsible for the software on the IoT device, except the firmware. The software vendor creates three sets of public-private key pairs: for OS packaging, the provisioning server, and IoT device (as a client). The public packaging key is used for verifying a ROMFS file. The keys for the provisioning server and client must be certified by the device supplier's CA. The expiration of the client certificate for an IoT device implies that the support for the second Linux has ended, and the IoT device cannot boot it any longer.

The provisioning server is used for setting up contents in the secure storage, i.e., the client private key, client public certificate, package public key, and the URL for the download server. The download server is managed by the service provider, which must set the URL and provide it to the software vendor.

5.3.5 Service Provider. The service provider is responsible for distributing the ROMFS file for the second Linux. It creates a key-pair for the TLS connection of the download server. The public key is

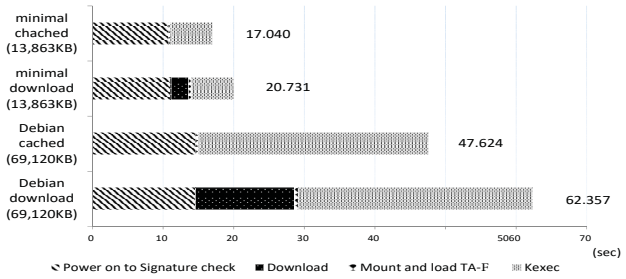

Figure 7: Elapsed time (in second) for each element to boot the second Linux (Minimal or Debian) with/without downloading.

used for the server certificate and certified by the device supplier's CA. The server certificate can be revoked when an emergency vulnerability is found.

6 EVALUATION

The software size, performance, and security are evaluated.

6.1 Size of RO-IoT

RO-IoT consists of two types of software: the secure network bootloader stored in the eMMC of Hikey board, and the second Linux in a downloaded ROMFS file. Their sizes affect hardware specification and performance for booting and memory forensics.

6.1.1 Size of Software on eMMC. The software used as a secure network bootloader is included in the "fip.bin" file and stored on the eMMC of the Hikey board as shown in Figure 3. The size of fip.bin is 7,590KB, which consists of Trusted Boot Firmware (29KB, BL2: Boot Loader 2), Secure Monitor (33KB, BL31), Secure OS OP-TEE (286KB, BL32), and the first Linux's ROMFS (7,100KB, BL33). The ROMFS file consists of the Linux kernel 4.15.0 (5,464KB), device tree file "dtb" (37KB), and initramfs.gz (1,598KB). The Linux kernel is minimized by the attack surface reduction tool "*undertaker*" [61]. The most part of initramfs.gz is occupied by TA-Boot (1,173KB), which includes general network tools (i.e., BoringSSL [22] and libWebSocket [63]).

The size of fib.bin is compared with Intel ME (Management Engine) which is based on MINIX3 kernel and offers the HTTPS service. The ME Cleaner project [72] shows the size of Intel ME (generation 3) is 7MB, which includes normal firmware. The fip.bin is almost the same size as the Intel ME, but it includes software to protect the normal Linux and the size is apparently acceptable.

6.1.2 Size of Downloaded ROMFS. We have prepared two images for the second Linux: a small-sized one (Minimal Linux) and another for easy-to-install (Debian Linux). Both images use the same kernel (4,960KB) and dtb (37KB), which are also minimized by the undertaker tool and removed some device drivers. Consequently, the total size difference comes from the divergence of the size of intramfs.gz and TA-Forensics. The Minimal Linux (13,863KB) includes initramfs.gz (8,637KB), and TA-Forensics (226KB). The Debian Linux (69,120KB) includes initramfs.gz (63,340KB), and TA-Forensics (781KB).

Table 2: Time for live memory forensics

simple-task	Scan Pages	Scan Time (sec)
0	2,591	0.296
100	4,324	1.566
200	5,997	2.862
300	7,687	4.117

The size of TA-Forensics depends on the size of the database for memory forensics. The Minimal Linux includes 64 ELF binaries and creates 2,375 SHA256-hashes (76KB) for the database in 226KB TA-Forensics. The Debian includes 788 ELF binaries and creates 11,796 SHA256-hashes (377KB) for the database in 781KB TA-Forensics. TA-Forensics contains the hash database, and the size becomes larger accordingly. The current OP-TEE passes TA-Forensics from the normal world to the secure world through shared memory (4MB), and TA-Forensics must be smaller than 4MB due to the OP-TEE implementation. Both TA-Forensics implementations are within the range.

As discussed the scalability in Section 4.2.4, 1TB traffic is estimated at 100 USD. When the size of Minimal and Debian Linux are 14MB and 70MB, the update traffic for 10,000 images are 0.14TB (14 USD) and 0.7TB (70 USD), respectively. The cost for a device is less than 1 cent. It is a simple estimation but shows the cost is inexpensive.

6.2 Performance

The time for booting and live memory forensics is measured for the two types of the second Linux.

6.2.1 Time for Booting. Figure 7 shows the elapsed time for each element to boot the second Linux. It illustrates the time until finishing the signature check of the cached ROMFS file, downloading, loopback-mounting, and booting the second Linux by *kexec*. The time until finishing the signature check is 10.8 seconds for 13,863KB Minimal Linux image and 14.4 seconds at 69,120KB Debian Linux image. The difference of approximately 4 seconds at finishing signature is due to the size difference of the second Linux in the eMMC.

The time for downloading the second Linux (i.e., ROMFS file) would be added when the signature check fails. The download time was 2.245 seconds for 13,863KB Minimal Linux image and 13.658 seconds for 69,120KB Debian image. They include time for world switch for every 128KB transfer, writing the copy of the image in memory for the next boot stage, and writing the download image to eMMC while encrypting it. The throughput was roughly 49Mbps and 40Mbps, respectively, which show good performance over the 100Mbps USB Ethernet.

The time from power on to finish network boot was 20.731 seconds and 62.357 seconds on Minimal and Debian, respectively. The availability was calculated to 99.986% and 99.958% if the system-reset is caused in 42 hours as mentioned in Section 4.3. If the cached images were valid, the boot times were 17.040 seconds and 47.624 seconds, and the availability was 99.988% and 99.968%, respectively. The results are apparently acceptable for replacing the whole OS image for a compromised system.

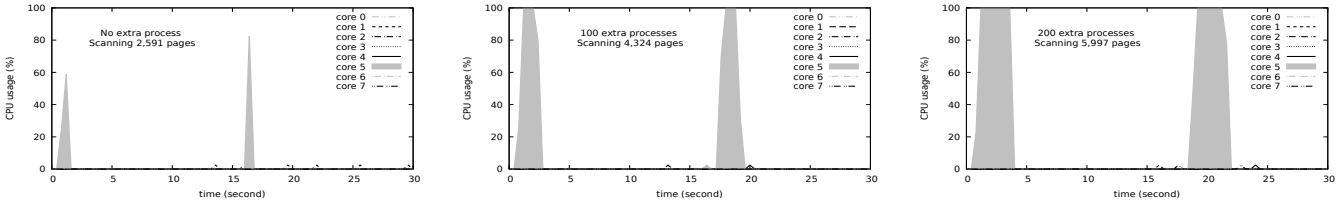

Figure 8: CPU utilization for memory forensics when 0, 100, and 200 "simple-task" processes run.

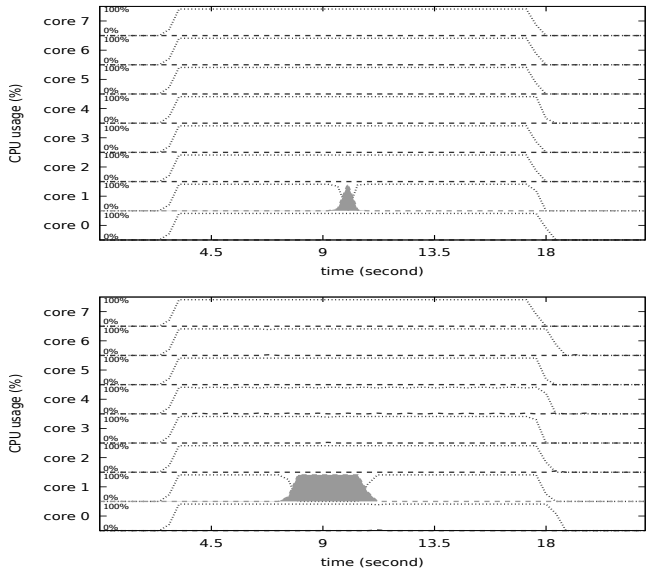

Figure 9: CPU utilization on each core with 8 heavy processes to measure the effect of memory forensics. The lower adds extra 200 "simple-task" processes. Small dotted lines show USER CPU utilization and dashed gray lines show SYSTEM CPU utilization.

6.2.2 Time for Live Memory Forensic. To confirm the performance of live memory forensics, a stress test is performed. An extra process (named "simple-task"), which adds two variables and sleeps 1 second in an infinite loop, is included in the Minimal Linux. The simple-task consumes memory pages and little CPU time. Figure 8 shows the CPU utilization when memory forensics occurred for 0, 100, and 200 simple-task processes.[2] When active processes were over 100, the CPU utilization went up to 100% on only one core, but the other 7 cores had no stress, implying that regular processes continue running without additional overhead.

Table 2 shows the scan time of live memory forensics for simple-task process numbers. The average times for memory forensics were 0.296, 1.566, 2.862, and 4.117 seconds at 0, 100, 200, and 300 simple-task processes, respectively. The evaluation shows simple-task increased about 17 pages per one simple-task. The scan time increased about 1.274 seconds when every 100 simple-task processes were added. The results suggest that approximately 1,177 simple-task processes become 20,000 pages, and it reaches 15 seconds to finish memory forensics. In this situation, the watchdog timer will not be extended, and a system-reset will be issued. We think the system is designed carefully, and this time-runs-out situation may not occur, but we discuss this problem in Section 7.2.

In addition, the impact of memory forensics was measured with 8 heavy processes which caused 100% CPU utilization on all 8 cores. The heavy process is designed to consume 15 seconds of CPU time to overlap the memory forensics. The memory forensics is a higher privilege and invoked during 8 processes. Figure 9 shows the impact. The upper case showed that memory forensics did not cause a big impact and the lower case added 200 simple-task processes that were executed to increase the overhead of memory forensics but little impact on the heavy processes. The 8 heavy processes were executed on 7 cores during the memory forensics ran. They were scheduled equally, and the finish time of each process was almost the same. The total time was delayed by the overhead of memory forensics and terminated after 18 seconds in Figure 9. This result also showed that *TEE_Spinlock(), which locked the task_struct*, did not affect the CPU performance.

6.3 Experimental Security Evaluation

We attempted to cause several security issues and experimentally confirmed that our current RO-IoT prototype can withstand such attacks. We describe a few examples here. RO-IoT live memory forensics process successfully detected an unregistered application in memory and issued a system-reset at the predefined interval. Note that this runtime integrity check and unknown binary detection differs from regular anti-virus malware detection (generally performed at download time or in the storage). In our case, a package manager (in this case Debian's *apt* command) could run and install applications, which RO-IoT could easily detect (as confirmed in our tests). Such an application was detected after it was loaded and executed, and the system-reset occurred because the binary hash of the application was not whitelisted in TA-Forensics.

The second Linux also could include an unregistered binary. However, TA-Forensics issued a system-reset when the binary was executed. Library replacement attack based on LD_PRELOAD was also detected, and the system-reset was issued duly.

We also tested performance of the security update process. Update of the firmware/bootloader (i.e., fip.bin 7.5MB) on the eMMC took about 2 seconds for downloading and 1.5 seconds for copying to eMMC (by dd command). The total time in our case, including

[2]Note: SMC instruction is executed on any core. The core number has no meaning in Figures 8 and 9.

the time until the signature check (see Section 6.2.1) was 10.8 and 14.4 seconds, for Minimal and Debian Linux, respectively.

7 SECURITY AND OTHER CONSIDERATIONS

In this section, we discuss several security and deployment issues related to RO-IoT, including: mission-critical applications, time span for live memory forensics and system-reset.

7.1 Self-Modifying Code and Swap

Our current memory forensics can detect the library replacement attacks that change *LD_ LIBRARY_PATH* and *LD_PRELOAD* environment variables (see Section 6.3). However, we do not support self-modifying code (including e.g., dl_open(), plug-ins, and Just-In-Time compiler), as such code is created dynamically, and we cannot pre-compute the hash values statically to check for unauthorized code. Although self-modifying code offers flexibility, it also increases the attack surface [21, 39]. Therefore, the use of self-modifying code should be refrained on special purpose systems, such as M2M IoT.

When a target memory page is swapped out to the storage, TA-Forensics cannot verify it. However, our second Linux excludes any storage driver, and consequently, no swap mechanism is available.

7.2 Time Span between Memory Forensics

The live memory forensics process may suffer from a *time-runs-out* problem, i.e., may fail to check all memory pages within a short period of time (before the next trigger of the watchdog timer). This can be easily mitigated by extending the watchdog timer period, albeit with the risk of enabling a larger time-window for malware to run. From the performance evaluation, our live memory forensics process can check 20,000 memory pages within 15 seconds. This seems to be adequate for many IoT applications.

The current time span is fixed to 15 seconds, but it can also be random. The random time span cannot be guessed by an attacker and makes it more difficult to abuse. To avoid the time-runs-out problem, the random time span must have a minimum duration.

7.3 Time Span between System-Reset

Emergency updates and recalls should be applied as soon as possible, but the automatic system-reset of RO-IoT is performed only occasionally on each device. Device termination caused by the life cycle management must wait for the next system-reset. Therefore, RO-IoT should not allow long time for the occasional system-reset. For our evaluation, we set the time span for 2 days, but it can be modified to suit the deployment needs for a target application.

Another concern for system-reset is the following: if many IoT devices reboot at the same time, this will break the assumption that the suspension time for rebooting on each device is hidden by AI-Edge applications. Therefore, each device should cause a system-reset randomly. The design of RO-IoT has no mechanism to adjust the system-reset, and each device causes a system-reset autonomously. Fortunately, the suspension time for rebooting is less than 60 seconds, and overlaps of suspensions seem to be rare because each device boots independently, and the boot time is short. However, the administrators must prevent aligning the system-reset on many devices. The random time span of memory forensics discussed in Section 7.2 could be one solution.

7.4 Reboot-time and Mission-critical IoT

The current RO-IoT prototype is assumed to run AI-Edge applications, which can accommodate the 60-second suspension time during rebooting. However, such a long reboot time may be a problem for some mission-critical applications (e.g., mobility system or life support equipment). However, rebooting (with the possibility of reinstallation) is essential for recovering from unknown attacks. To mitigate the reboot time, the use of fault-tolerance technology can be considered; e.g., see [4, 49] for examples of how fault-tolerance and rebooting can coexist, in special conditions such as a helicopter controlled by real time OS.

On the other hand, OS update without complete rebooting is a long standing research topic [12, 41, 53, 109]. However, our current RO-IoT design focuses on preventing long-lived malware (e.g., rootkit), and adopts a full system-reset. Existing techniques for OS updating without a complete reboot should be examined in future work to make RO-IoT amenable to mission-critical IoT applications.

8 CONCLUSION

We propose the Reboot-Oriented IoT as a security mechanism for M2M IoT devices, especially for AI-Edge applications. RO-IoT causes a system-reset proactively in the TEE (namely, the ARM TrustZone), which enables recovery from a hijacked/compromised device. The rebooting process is also protected by TEE, which verifies the life cycles of device, software, and service components of the IoT device, which are linked to PKI-based certificates of CA, server, and client. When one of the certificates expires, the device becomes inoperable, allowing strict life-cycle management, and facilitating product recalls, if necessary. The data stored in the IoT device is encrypted by a hardware-protected SoC key, and therefore the device is disposable. While the normal OS is running, RO-IoT repeats a memory scan to enforce application whitelisting and the integrity of the allowed applications. Our current prototype is carefully implemented to avoid known TrustZone vulnerabilities, such as the boomerang attack and replay attack. The implementation on the ARM Hikey board shows a reasonable storage size (8MB bootloader protected by TrustZone), performance (14MB Linux boots within 21 seconds from the network or 17 seconds from cache), scalability (less than 1 cent for full OS update per device), and availability (over 99.9%).

ACKNOWLEDGMENTS

We are grateful to our shepherd Hussain Almohri for guiding us in the final version of this paper. We also thank the ACSAC2020 anonymous reviewers for their insightful suggestions and comments.

REFERENCES

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 variability bugs in the Linux kernel: a qualitative analysis. In *International Conference on Automated Software Engineering (ASE)*.

[2] Fardin Abdi, Chien-Ying Chen, Monowar Hasan, Songran Liu, Sibin Mohan, and Marco Caccamo. 2018. Guaranteed Physical Security with Restart-Based Design for Cyber-Physical Systems. In *International Conference on Cyber-Physical Systems (ICCPS)*.

[3] Fardin Abdi, Monowar Hasan, Sibin Mohan, Disha Agarwal, and Marco Caccamo. 2016. ReSecure: A Restart-Based Security Protocol for Tightly Actuated Hard Real-Time Systems. In *IEEE Workshop on Security and Dependability of Critical Embedded Real-Time Systems (CERTS)*.

[4] Fardin Abdi, Rohan Tabish, Matthias Rungger, Majid Zamani, and Marco Caccamo. 2017. Application and System-Level Software Fault Tolerance through Full System Restarts. In *International Conference on Cyber-Physical Systems (ICCPS)*.

[5] Murat Ali, Jozef Hubertus Alfonsus Vlaskamp, Nof Nasser Eddin, Ben Falconer, and Colin Oram. 2013. Technical Development and Socioeconomic Implications of the Raspberry Pi as a Learning Tool in Developing Countries. In *Computer Science and Electronic Engineering Conference (CEEC)*.

[6] Werner Almesberger. 2006. kboot - A boot loader based on Kexec. In *Proceedings of the Linux symposium (OLS)*.

[7] Hussain MJ Almohri, Layne T Watson, and David Evans. 2017. Misery Digraphs: delaying intrusion attacks in obscure clouds. *IEEE Transactions on Information Forensics and Security* 13, 6 (2017), 1361–1375.

[8] Amazon Web Services. 2019. *Amazon EC2 Pricing, https://aws.amazon.com/ec2/pricing/on-demand/*.

[9] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Seaman Chad, Nick Sullivan, Kurt Thomas, and Yi Zhou. 2017. Understanding the Mirai Botnet. In *USENIX Security symposium*.

[10] ARE Project. 2018. *http://are1.tech/*.

[11] Joe Armstrong. 2003. *Making reliable distributed systems in the presence of software errors*. Ph.D. Dissertation. Mikroelektronik och informationsteknik.

[12] Jeff Arnold and M Frans Kaashoek. 2009. Ksplice: Automatic Rebootless Kernel Updates. In *European conference on Computer systems, (EuroSys)*.

[13] Miguel A Arroyo, M Tarek Ibn Ziad, Hidenori Kobayashi, Junfeng Yang, and Simha Sethumadhavan. 2019. YOLO: Frequently Resetting Cyber-Physical Systems for Security. In *Autonomous Systems: Sensors, Processing, and Security for Vehicles and Infrastructure 2019*, Vol. 11009.

[14] Tarek M. Attia. 2019. *Challenges and Opportunities in the Future Applications of IoT Technology*. https://www.econstor.eu/bitstream/10419/201752/1/ITS2019-Aswan-paper-61.pdf

[15] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Computer and Communications Security (CCS)*.

[16] Ahmed M Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning. 2016. SKEE: A lightweight Secure Kernel-level Execution Environment for ARM. In *Network and Distributed System Security Symposium (NDSS)*.

[17] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. 2015. A Survey on Hypervisor-Based Monitoring: Approaches, Applications, and Evolutions. *ACM Computing Surveys (CSUR)* 48, 1 (2015), 1–33.

[18] BBC News. 2019. *Children's smartwatch recalled over data fears, https://www.bbc.com/news/technology-47130269*.

[19] Elisa Bertino and Nayeem Islam. 2017. Botnets and Internet of Things Security. *Computer* 50, 2 (2017), 76–79.

[20] Manish Bhatt, Irfan Ahmed, and Zhiqiang Lin. [n.d.]. Using Virtual Machine Introspection for Operating Systems Security Education. In *ACM Technical Symposium on Computer Science Education*.

[21] Dion Blazakis. 2010. Interpreter exploitation: Pointer inference and JIT spraying. In *Black Hat DC*.

[22] BoringSSL. 2014. *https://boringssl.googlesource.com/boringssl/*.

[23] Benjamin Bucklin Brown. 2018. Over-the-Air (OTA) Updates in Embedded Microcontroller Applications: Design TradeOffs and Lessons Learned. *Analog Dialogue Technical Journal* 52 (2018), 52–11.

[24] Seraphin B Calo, Maroun Touna, Dinesh C Verma, and Alan Cullen. 2017. Edge Computing Architecture for applying AI to IoT. In *IEEE International Conference on Big Data (Big Data)*.

[25] George Candea and Armando Fox. 2003. Crash-Only Software. In *Hot Topics in Operating Systems (HotOS)*.

[26] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. 2003. Microreboot–A Technique for Cheap Recovery. In *USENIX Annual Technical Conference (USENIX ATC)*.

[27] Censys. 2016. *https://censys.io/*.

[28] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. 2020. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *IEEE Symposium on Security and Privacy (IEEE S&P)*.

[29] Ramesh Chandra, Nickolai Zeldovich, Constantine Sapuntzakis, and Monica S. Lam. 2005. The Collective: A Cache-based System Management Architecture. In *Networked Systems Design & Implementation (NSDI)*.

[30] Yue Chen, Yulong Zhang, Zhi Wang, and Tao Wei. 2017. Downgrade Attack on TrustZone. *arXiv*.

[31] Suranjan Choudhury, Kartik Bhatnagar, and Wasim Haque. 2002. *Public key infrastructure implementation and design*. John Wiley & Sons, Inc.

[32] Kees Cook. 2017. Linux Kernel Self-Protection. *; login:* 42, 1 (2017), 14–17.

[33] Kees Cook. 2018. The State of Kernel Self Protection. In *Linux Conf AU*.

[34] Sam Edwards and Ioannis Profetis. 2016. Hajime: Analysis of a decentralized internet worm for IoT devices. *Rapidity Networks* 16 (2016).

[35] Carl Ellison and Bruce Schneier. 2000. Ten Risks of PKI: What you are not being told about Public Key Infrastructure. *Computer security journal* 16, 1 (2000), 1–7.

[36] Xuan Feng, Qiang Li, Haining Wang, and Limin Sun. 2018. Acquisitional Rule-based Engine for Discovering Internet-of-Thing Devices. In *USENIX Security symposium*.

[37] Alissa M Fitzgerald. 2018. The Internet of disposable things: Throwaway paper and plastic sensors will connect everyday items. *IEEE Spectrum* 55, 12 (2018), 30–35.

[38] Yangchun Fu and Zhiqiang Lin. 2012. Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *2012 IEEE symposium on security and privacy (IEEE SP)*.

[39] Robert Gawlik and Thorsten Holz. 2018. SoK: Make JIT-Spray Great Again. In *USENIX Workshop on Offensive Technologies (WOOT)*.

[40] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. 2014. Sprobes: Enforcing Kernel Code Integrity on the TrustZone Architecture. *Mobile Security Technology Workshop (MoST)* (2014).

[41] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. 2013. Safe and Automatic Live Update for Operating Systems. *ACM SIGPLAN Notices* 48, 4 (2013), 279–292.

[42] Google Cloud Platform. 2019. *Network Pricing, https://cloud.google.com/compute/network-pricing*.

[43] Mariano Graziano, Andrea Lanzi, and Davide Balzarotti. 2013. Hypervisor Memory Forensics. In *Recent Advances in Intrusion Detection (RAID)*.

[44] Russ Housley and Tim Polk. 2001. *Planning for PKI: best practices guide for deploying public key infrastructure*. John Wiley & Sons, Inc.

[45] IEEE Internet Technology Policy Community. 2017. Internet of Things (IoT) security best practices. In *IEEE Internet Technology Policy Community White Paper*.

[46] iPXE. 2010. *https://ipxe.org/*.

[47] IT Cornell. 2018. *Recover From a System Compromise, https://it.cornell.edu/security-essentials-it-professionals/recover-system-compromise*.

[48] ITU. 2019. *Handbook for the development of a policy framework on ICT/e-waste*. https://www.itu.int/en/ITU-D/Climate-Change/Documents/2018/Handbook-Policy-framework-on-ICT-Ewaste.pdf

[49] Pushpak Jagtap, Fardin Abdi, Matthias Rungger, Majid Zamani, and Marco Caccamo. 2020. Software Fault Tolerance for Cyber-Physical Systems via Full System Restart. *ACM Transactions on Cyber-Physical Systems* 4, 4 (2020), 1–20.

[50] Jin Soo Jang, Sunjune Kong, Minsu Kim, Daegyeong Kim, and Brent Byunghoon Kang. 2015. SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment. In *Network and Distributed System Security Symposium (NDSS)*.

[51] Yujuan Jiang, Bram Adams, and Daniel M German. 2013. Will my patch make it? And how fast? Case study on the Linux kernel. In *Working Conference on Mining Software Repositories*.

[52] Stephen T Jones, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2006. Antfarm: Tracking Processes in a Virtual Machine Environment. In *USENIX Annual Technical Conference (USENIX ATC)*.

[53] Sanidhya Kashyap, Changwoo Min, Byoungyoung Lee, Taesoo Kim, and Pavel Emelyanov. 2016. Instant OS Updates via Userspace Checkpoint-and-Restart. In *USENIX Annual Technical Conference (USENIX-ATC)*.

[54] Kernel Self-Protection. 2019. *https://www.kernel.org/doc/html/v5.4/security/self-protection.html*.

[55] Samuel T King and Peter M Chen. 2006. SubVirt: Implementing malware with virtual machines. In *IEEE Symposium on Security and Privacy (IEEE SP)*.

[56] Ryozo Kiyohara, Satoshi Mii, Mitsuhiro Matsumoto, Masayuki Numao, and Satoshi Kurihara. 2009. A new method of fast compression of program code for OTA updates in consumer devices. *IEEE Transactions on Consumer Electronics* 55, 2 (2009), 812–817.

[57] Andi Kleen. 2013. gcc link time optimization and the Linux kernel. In *Linux Collab Summit*.

[58] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. In *arXiv*.

[59] Paul Kocher, J. Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, M. Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (IEEE SP)*.

[60] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. 2017. DDoS in the IoT: Mirai and Other Botnets. *IEEE Computer* 50, 7 (2017), 80–84.

[61] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. 2013. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *Network and Distributed System Security Symposium (NDSS)*.

[62] Rongfeng Li, Liu Wang, Deying Kong, and Lan Yin. 2018. Recent progress on biodegradable materials and transient electronics. *Bioactive materials* 3, 3 (2018), 322–333.

[63] LibWebSocket. 2013. *https://libwebsockets.org/*.

[64] Linaro. 2020. *OP-TEE Documentation.* https://readthedocs.org/projects/optee/downloads/pdf/latest/

[65] Linux Tinification. 2014. *https://tiny.wiki.kernel.org*.

[66] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. 2018. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*.

[67] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. In *arXiv*.

[68] Aravind Machiry, Eric Gustafson, Chad Spensky, Chris Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. 2017. Boomerang: Exploiting the semantic gap in trusted execution environments. In *Network and Distributed System Security Symposium (NDSS)*.

[69] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya. 2018. Fog computing: A taxonomy, survey and future directions. In *Internet of everything*. Springer, 103–130.

[70] Artur Marzano, David Alexander, Osvaldo Fonseca, Elverton Fazzion, Cristine Hoepers, Klaus Steding-Jessen, Marcelo HPC Chaves, Ítalo Cunha, Dorgival Guedes, and Wagner Meira. 2018. The Evolution of Bashlite and Mirai IoT Botnets. In *IEEE Symposium on Computers and Communications (ISCC)*.

[71] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. In *USENIX Security Symposium*.

[72] ME Cleaner Project. 2017. *https://github.com/corna/me_cleaner/*.

[73] Yasir Mehmood, Farhan Ahmad, Ibrar Yaqoob, Asma Adnane, Muhammad Imran, and Sghaier Guizani. 2017. Internet-of-Things-Based Smart Cities: Recent Advances and Challenges. *IEEE Communications Magazine* 55, 9 (2017), 16–24.

[74] Mitre Challege IoT. 2017. *https://www.mitre.org/research/mitre-challenge/mitre-challenge-iot*.

[75] Vivek Mohan. 2018. Disposable IoT ready to open new opportunities, https://www.networkworld.com/article/3262970/disposable-iot-ready-to-open-new-opportunities.html. *NETWORK WORLD* (2018).

[76] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. 2016. TrustZone Explained: Architectural Features and Use Cases. In *International Conference on Collaboration and Internet Computing (CIC)*.

[77] Dennis K Nilsson and Ulf E Larson. 2008. Secure Firmware Updates over the Air in Intelligent Vehicles. In *IEEE International Conference on Communications Workshops (ICC)*.

[78] Zhenyu Ning and Fengwei Zhang. 2017. Ninja: Towards Transparent Tracing and Debugging on ARM. In *USENIX Security symposium*.

[79] OP-TEE. 2016. *https://www.op-tee.org/*.

[80] Michael Opdenacker. 2017. Embedded Linux size reduction techniques. In *Embedded Linux Conference (ELC)*.

[81] Himanshu Pareek, Sandeep Romana, and PRL Eswari. 2012. Application whitelisting: approaches and challenges. *International Journal of Computer Science, Engineering and Information Technology (IJCSEIT)* 2, 5 (2012), 13–18.

[82] Heejin Park, Shuang Zhai, Long Lu, and Felix Xiaozhu Lin. 2019. Streambox-TZ: secure stream analytics at the edge with trustzone. In *USENIX Annual Technical Conference (USENIX ATC)*.

[83] Charith Perera, Yongrui Qin, Julio C Estrella, Stephan Reiff-Marganiec, and Athanasios V Vasilakos. 2017. Fog Computing for Sustainable Smart Cities: A Survey. *ACM Computing Surveys (CSUR)* 50, 3 (2017), 1–43.

[84] Andy Pfiffer. 2003. Reducing System Reboot Time With kexec. *OSDL Whitepaper* (2003).

[85] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 1–36.

[86] Alessandro Reina, Aristide Fattori, Fabio Pagani, Lorenzo Cavallaro, and Danilo Bruschi. 2012. When Hardware Meets Software: A Bulletproof Solution to Forensic Memory Acquisition. In *Annual Computer Security Applications Conference (ACSAC)*.

[87] Ju Ren, Hui Guo, Chugui Xu, and Yaoxue Zhang. 2017. Serving at the Edge: A Scalable IoT Architecture Based on Transparent Computing. *IEEE Network* 31, 5 (2017), 96–105.

[88] Reuters. 2019. *China's Xiongmai to recall up to 10,000 webcams after hack, https://www.reuters.com/article/us-cyber-attacks-china/chinas-xiongmai-to-recall-up-to-10000-webcams-after-hack-idUSKCN12P1TT*.

[89] Joanna Rutkowska. 2006. Subverting VistaTM Kernel For Fun And Profit. *Black Hat USA*.

[90] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. 2015. Trusted Execution Environment: What It is, and What It is Not. In *2015 IEEE Trustcom/BigDataSE/ISPA*.

[91] Adam Sedgewick, Murugiah Souppaya, and Karen Scarfone. 2015. Guide to application whitelisting. *NIST Special Publication* 800 (2015).

[92] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE internet of things journal* 3, 5 (2016), 637–646.

[93] Shodan. 2013. *https://www.shodan.io/*.

[94] Bill Stearns. 2020. *Making a Solar Powered Raspberry Pi.* https://www.activecountermeasures.com/making-a-solar-powered-raspberry-pi/

[95] Kehua Su, Jie Li, and Hongbo Fu. 2011. Smart City and the Applications. In *2011 international conference on electronics, communications and control (ICECC)*.

[96] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Sushil Jajodia. 2014. TrustDump: Reliable Memory Acquisition on Smartphones. In *European Symposium on Research in Computer Security (ESORICS)*.

[97] TCG. 2019. *TPM 2.0 Authenticated Countdown Timer (ACT) Command.* https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM_ACTCommand_v1r3_pubrev.pdf

[98] Sonam Tenzin, Satetha Siyang, Theerapat Pobkrut, and Teerakiat Kerdcharoen. 2017. Low Cost Weather Station for Climate-Smart Agriculture. In *2017 9th international conference on knowledge and smart technology (KST)*.

[99] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying Linux Bug Fixing Patches. In *International Conference on Software Engineering (ICSE)*.

[100] UC Berkeley Information Security and Policy. 2018. *Reinstalling Your Compromised Computer, https://security.berkeley.edu/resources/best-practices-how-articles/compromised-systems/reinstalling-your-compromised-computer*.

[101] UCL Information Security Group. 2013. *Recovering from an intrusion, https://www.ucl.ac.uk/informationsecurity/itsecurity/knowledgebase/ securitybaselines/recovering*.

[102] Jiang Wang, Angelos Stavrou, and Anup Ghosh. 2010. HyperCheck: A Hardware-Assisted Integrity Monitor. In *Recent Advances in Intrusion Detection (RAID)*.

[103] Xueqiang Wang, Yuqiong Sun, Susanta Nanda, and XiaoFeng Wang. 2019. Looking from the Mirror: Evaluating IoT Device Security through Mobile Companion Apps. In *USENIX Security Symposium*.

[104] Sjaak Wolfert, Lan Ge, Cor Verdouw, and Marc-Jeroen Bogaardt. 2017. Big Data in Smart Farming–A Review. *Agricultural Systems* 153 (2017), 69–80.

[105] Guanping Xiao, Zheng Zheng, Bo Jiang, and Yulei Sui. 2019. An Empirical Study of Regression Bug Chains in Linux. *IEEE Transactions on Reliability* 69, 2 (2019), 558–570.

[106] Meng Xu, Manuel Huber, Zhichuang Sun, Paul England, Marcus Peinado, Sangho Lee, Andrey Marochko, Dennis Mattoon, Rob Spiger, and Stefan Thom. 2019. Dominance as a New Trusted Computing Primitive for the Internet of Things. In *IEEE Symposium on Security and Privacy (IEEE SP)*.

[107] Fengwei Zhang, Jiang Wang, Kun Sun, and Angelos Stavrou. 2013. HyperCheck: A Hardware-Assisted Integrity Monitor. *IEEE Transactions on Dependable and Secure Computing* 11, 4 (2013), 332–344.

[108] Zhi-Kai Zhang, Michael Cheng Yi Cho, Chia-Wei Wang, Chia-Wei Hsu, Chong-Kuan Chen, and Shiuhpyng Shieh. 2014. IoT Security: Ongoing Challenges and Research Opportunities. In *International Conference on Service-Oriented Computing and Applications (SOCA)*.

[109] Lei Zhou, Fengwei Zhang, Jinghui Liao, Zhengyu Ning, Jidong Xiao, Kevin Leach, Westley Weimer, and Guojun Wang. 2020. KShot: Live Kernel Patching with SMM and SGX. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[110] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. 2019. Edge Intelligence: Paving the Last Mile of Artificial Intelligence With Edge Computing. *Proc. IEEE* 107, 8 (2019), 1738–1762.