Supporting Transparent Snapshot for Bare-metal Malware Analysis on Mobile Devices

Le Guan
Pennsylvania State University, USA

Shijie Jia*†
Data Assurance and Communication Security Research Center, CAS, China

Bo Chen
Michigan Technological University, USA

Fengwei Zhang
Wayne State University, USA

Bo Luo
The University of Kansas, USA

Jingqiang Lin‡
Data Assurance and Communication Security Research Center, CAS, China

Peng Liu
Pennsylvania State University, USA

Xinyu Xing
Pennsylvania State University, USA

Luning Xia§
Data Assurance and Communication Security Research Center, CAS, China

ABSTRACT

The increasing growth of cybercrimes targeting mobile devices urges an efficient malware analysis platform. With the emergence of evasive malware, which is capable of detecting that it is being analyzed in virtualized environments, bare-metal analysis has become the definitive resort. Existing works mainly focus on extracting the malicious behaviors exposed during bare-metal analysis. However, after malware analysis, it is equally important to quickly restore the system to a clean state to examine the next sample. Unfortunately, state-of-the-art solutions on mobile platforms can only restore the disk, and require a time-consuming system reboot. In addition, all of the existing works require some in-guest components to assist the restoration. Therefore, a kernel-level malware is still able to detect the presence of the in-guest components.

We propose `Bolt`, a transparent restoration mechanism for bare-metal analysis on mobile platform without rebooting. `Bolt` achieves a reboot-less restoration by simultaneously making a snapshot for both the physical memory and the disk. Memory snapshot is enabled by an isolated operating system (`BoltOS`) in the ARM TrustZone secure world, and disk snapshot is accomplished by a piece of customized firmware (`BoltFTL`) for flash-based block devices. Because both the `BoltOS` and the `BoltFTL` are isolated from the guest system, even kernel-level malware cannot interfere with the restoration. More importantly, `Bolt` does not require any modifications into the guest system. As such, `Bolt` is the first that simultaneously achieves efficiency, isolation, and stealthiness to recover from infection due to malware execution. We have implemented a `Bolt` prototype working with the Android OS. Experimental results show that `Bolt` can restore the guest system to a clean state in only 2.80 seconds.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; *Hardware security implementation*;

KEYWORDS

Bare-metal Analysis, Evasive Malware, Snapshot, Flash-based Block Device

ACM Reference Format:
Le Guan, Shijie Jia, Bo Chen, Fengwei Zhang, Bo Luo, Jingqiang Lin, Peng Liu, Xinyu Xing, and Luning Xia. 2017. Supporting Transparent Snapshot for Bare-metal Malware Analysis on Mobile Devices. In *Proceedings of ACSAC 2017, Orlando, FL, USA, December 4–8, 2017,* 11 pages.
https://doi.org/10.1145/3134600.3134647

1 INTRODUCTION

Smartphones are increasingly becoming the targets of cybercrimes. Therefore, detection of malicious behaviors beforehand is a top priority for multiple stakeholders, such as the hardware/software manufactures and end users. Although conventional static analysis and signature-based detection mechanisms have been effective, they still cannot detect all types of malware. In particular, when techniques such as obfuscation, packing, and polymorphism are employed by the malware, static analysis methods fall short. Dynamic analysis techniques overcome these limitations by executing the samples in a sandboxed environment [10, 11, 14, 17, 32, 41, 48, 51]. The malicious behaviors can be captured by the in-guest[1] monitoring components (e.g., using `ptrace`) or lower-level out-of-guest monitoring components (e.g., a security monitor in VMM). Unfortunately, more sophisticated malware or evasive malware is capable of detecting the existence of the analytic components or

*Corresponding author.
†Also with State Key Laboratory of Information Security, IIE, CAS, China.
‡Also with State Key Laboratory of Information Security, IIE, CAS, China.
§Also with State Key Laboratory of Information Security, IIE, CAS, China.

ACSAC 2017, December 4–8, 2017, Orlando, FL, USA

ACM ISBN 978-1-4503-5345-8/17/12…$15.00
https://doi.org/10.1145/3134600.3134647

[1]In this paper, as a convention, "guest" means the system in which the malware runs.

the emulated/virtualized execution environment, and refrains from exposing any malicious activity [5, 7, 13, 23, 36].

In recent years, the bare-metal dynamic analysis technique is becoming popular [1, 24–26, 30, 31, 40, 49, 50, 53]. By executing the malware in an unmodified Operating System (OS) installation that runs on an actual hardware, the malware cannot identify the existence of the monitoring component. Hence, the malware considers itself running on a victim device, and starts to reveal its malicious behaviors. There are two key challenges faced by bare-metal dynamic analysis technique. First, without an auxiliary component in the guest system (or a security monitor in VMM), it is difficult to transparently and accurately collect malware behaviors. Second, unlike virtualization based solutions, restoring the analyzing system to a clean state after each malware infection usually requires a system reboot, which is inherently slow. However, the ever-increasing flood of new malware demands malware analysis solutions with high performance and high throughput. To automate the analysis in a scalable way, the time for system restoration must be minimized.

While many efforts have been devoted to the first aspect, i.e., behavior extraction [1, 24, 26, 31, 40, 50, 53], very little attention has been paid to the aspect of quick restoration. State-of-the-art solutions either require modification to the guest system, thus breaking the transparency to malware [25, 30], or only restore part of the system state [30]. In particular, by employing a dedicated small OS, BareBox [25] is able to restore the system state, including both memory and disk contents, within four seconds. However, both the small OS and the in-guest component to assist restoration are running at the same privilege level as the guest system. Therefore, a kernel-level malware is able to identify that it is being analyzed, and even disrupt the small OS. BareDroid [30] is specifically designed for the quick restoration of the Android OS. However, it only handles disk image, and hence requires a full reboot after each malware infection. Moreover, it cannot eliminate the existence of in-guest components either.

In this paper, we propose `Bolt`, an OS-agnostic mechanism which can quickly restore an ARM-based device to a designated state. `Bolt` is able to make a full system snapshot (including both the physical memory and the disk storage) at anytime during execution, and quickly restore the system state without rebooting. Compared to existing work [25, 30], `Bolt` supports unmodified OS on the bare-metal platform, hence the fingerprint of its presence is minimized. In addition, `Bolt` achieves three goals simultaneously: (1) **Stealthiness**. The component performing the restoration is transparent to the guest OS that runs the malware. (2) **Isolation**. The malware, even with kernel privilege, cannot interfere with the restoration component. (3) **Efficiency**. `Bolt` can directly restore the system to a clean checkpoint, eliminating the need of rebooting.

To achieve the design goals, we novelly utilize two hardware features – ARM TrustZone security extension and flash-based storage device, both of which are standard hardware features in all the mainstream ARM-based mobile platforms. ARM TrustZone extends ARM processors to provide two virtual processor cores that run with different privilege levels. Normal mobile OSes, such as Android and iOS, run in the normal world with less privileges, while a lightweight OS in the secure world runs trusted applications (e.g., Trustlets) that provides security-critical services, such as fingerprint recognition to the normal OS. Another hardware feature equipped with mobile devices is flash-based storage device. Nowadays, flash memory in the forms of eMMC cards, SD cards, MicroSD cards are pervasively used in mobile platforms due to its high throughput, low energy consumption, and small size.

Leveraging TrustZone and flash memory, `Bolt` performs whole-system (including both memory state and disk state) recovery without rebooting in the TrustZone secure world: (1) For the *memory state*, `Bolt` partitions the physical memory into three regions – one for the secure-world trusted OS (i.e., `BoltOS`), one for the normal-world guest OS, and the other for the snapshot of the guest OS. `BoltOS` runs in the secure world, and is responsible for saving/restoring the physical memory of the guest OS to/from the snapshot region. Note that the snapshot can only be accessed by `BoltOS` in the secure world. (2) For the *disk state*, `Bolt` takes advantage of a special design feature in flash-based block devices – out-of-place update. To accommodate the unique nature of flash memory (Section 2.3), data overwriting in flash is usually performed "out of place", in which the new data are always stored in the newly allocated flash space, while the old data being over-written will remain intact until garbage collection is performed. Utilizing this feature, we are able to restore flash state by only backing up a small amount of metadata rather than the entire data. This is because the OS can only access the flash storage via the interface provided by flash firmware, with which we can simply manipulate the OS view of data so that the clean-state data is invisible to the malware. Finally, we carefully modify garbage collection/wear leveling to prevent them from damaging the clean-state data.

Since `BoltOS` works without assistance from the guest OS, it is stealthy to kernel-level malware. Moreover, with TrustZone, `BoltOS` can be isolated from the guest OS. Therefore, kernel-level malware cannot disrupt it. Finally, the performance of `Bolt` is greatly boosted by avoiding both system reboot and a full restoration of flash data. In summary, our work makes the following main contributions:

- We propose `Bolt`, a novel approach that could transparently restore the whole-system state of a running OS without reboot. To the best of our knowledge, `Bolt` is the first design which can simultaneously achieve stealthiness, isolation and efficiency.
- We design a new algorithm for flash-based block storage that supports hardware-based checkpoint working in conjunction with `BoltOS`.
- As a proof of concept, we implement `Bolt` using the Android OS as the guest system.[2] Experimental results show that the proposed system is able to make a full system recovery in 2.80 seconds, significantly outperforming state-of-the-art solutions.

Roadmap. The rest of this paper is structured as follows. In Section 2, we discuss necessary background information on Android, ARM TrustZone and flash-based block storage. Section 3 and Section 4 describe `Bolt` design and implementation respectively. Then we evaluate the performance of the proposed system in Section 5, followed by a discussion of the drawbacks of the current prototype

[2] Although our system is OS-agnostic, we cannot do experiment with iOS, because iOS devices are tightly controlled in hardware.

and future work in Section 6. We review related work in Section 7. Finally, Section 8 concludes the paper.

2 BACKGROUND

This section presents necessary background information of the proposed system. We first briefly introduce the design of the Android OS and its partitions. We then describe ARM TrustZone, a popular security extension to the ARM processors. This security feature is the root of trust in most mobile devices in the market. Finally, we introduce flash-based block devices, the most popular forms of storage media used in mobile devices.

2.1 Android

Android is an open-source OS based on Linux kernel. It is specifically optimized for mobile devices like smart phones and tablets. Each Android application (i.e., app) is assigned a unique user ID, and runs in its own virtual machine (VM) instance, which is sandboxed to avoid accessing the rest of the system. The apps are written in the Java programming language, which are first compiled to bytecode for the Java virtual machine, and then translated to a customized Dalvik Executable (DEX) format. The app package is loaded into the address space of a VM, along with framework libraries that can be invoked to request for system services.

Android partitions. The Android system organizes its non-volatile storage into several partitions: /boot, /recovery, /system, /data, /cache, /misc, etc. Each partition plays a distinct role and facilitates different functionality of the device. We briefly describe a few important partitions in the following:

- **Bootloader**: A non-filesystem partition that takes over the system after executing the proprietary device ROM. It initializes the SoC components such as DRAM, and then copies the kernel and ramdisk from the `Boot` partition into the memory.
- **Boot**: A non-filesystem partition flashed with an image containing the kernel and the ramdisk.
- **Recovery**: An alternative boot partition which serves for advanced recovery and maintenance operations.
- **System**: This partition contains the entire operating system except the kernel and the ramdisk. This includes the entire Android framework and the pre-installed system apps, such as the telephony.
- **Data**: This partition contains user data, e.g., user contacts, messages and user installed apps. Assuming that the `system` partition is unmodified, wiping this partition essentially performs a factory reset.

As a reboot-less system, `Bolt` needs to restore the non-volatile storage to a clean state, which only requires recovering the content in the `System` and `Data` partitions.

2.2 ARM TrustZone and Its Usage in Android

TrustZone is a security extension added to the ARM architecture [2]. It provides an isolated execution environment for sensitive tasks. In particular, it introduces two worlds, a *normal world* and a *secure world*. The security-critical workloads run in the isolated secure world, while the commodity OS runs in the normal world. Tasks running in different worlds have different privileges to access system resources. The secure-world components are allowed to access all the resources system-wide, but the normal-world components are only allowed to access non-secure resources. The current execution environment is determined by the *NS* (non-secure) bit in the *Security Configuration Register (SCR)*, which can only be accessed when the processor runs in the secure world. When this bit is set, the processor is in the normal world. Otherwise, the processor is in the secure world. To switch to the other world, the privileged code needs to issue an `SMC` instruction, which traps the processor in the *Monitor* mode. In this mode, the processor has ultimate privilege. Apart from accessing secure resources, it is able to manipulate the registers in both worlds. As a result, the *Monitor* mode serves as the gate for world-switching.

One of the most important components in a TrustZone-based system is the *TrustZone Address Space Controller (TZASC)*. Registers of *TZASC* are mapped into the physical address of the SoC, and can be accessed via normal memory operations. By configuring *TZASC* in the secure world, the physical memory can be split into several regions with different security levels. With these configurations, secure world software can control whether a memory region can be accessed in both the secure and normal worlds, or can only be accessed in the secure world.

Newer Android systems comprise of two parts residing in two worlds. The normal Android OS, including a customized Linux kernel, framework libraries, and apps runs in the normal world. A lightweight trusted OS runs in the secure world and provides security-critical services to the normal Android OS. The secure-world OS essentially constructs a *Trusted Execution Environment (TEE)* for running trusted services. To request a security-critical service, the normal Android OS issues an `SMC` instruction, which traps the processor into the secure world. Based on the contents of registers or shared memory, the trusted OS is able to identify the intent of the request, and schedule a piece of trusted code (called Trustlet) to perform the actual computation. Finally, the results are sent back to the normal-world Android OS.

2.3 Flash-based Block Devices

Flash-based block devices (e.g., eMMC cards, SD cards, MicroSD cards and SSD drives) have been widely used to replace the conventional hard disk drives (HDD) due to their high I/O performance and low energy consumption. Particularly, popular flash products like eMMC cards and SD cards have dominated the storage media of mobile devices (e.g., smart phones, tablets, smart watches).

Flash memory. Flash memory is a non-volatile storage medium which can be electrically erased and reprogrammed. The flash memory family contains NOR-type and NAND-type flash. The NOR flash allows one-byte random access, and is usually expensive with relatively small capacity. Thus, it is commonly used to store executable program (e.g., bootloader). The NAND flash, however, is cheaper and has much larger capacity, thus it has been pervasively used in flash storage media. NAND flash stores information in an array of memory cells, which are grouped into blocks. The size of a block is usually a few hundred Kilobytes. A flash block is further divided into pages, each of which can be 512 bytes, 2 KB, or 4 KB. Note that

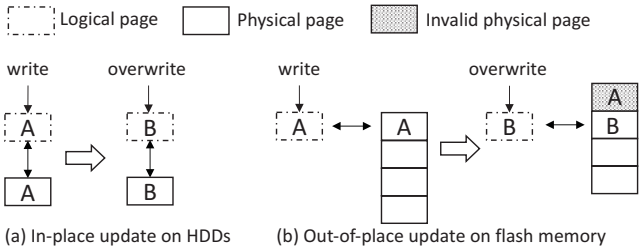


Figure 1: Comparison of an overwrite operation between HDD and flash memory.

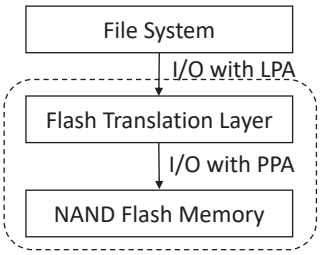


Figure 2: The architecture of a flash-based storage system using FTL. LPA: logical page addresses, PPA: physical page addresses.

reading/writing flash is usually performed on the basis of pages, and erasure can only be performed on a block basis.

Compared to conventional mechanical HDDs, flash memory has completely different characteristics. First, a flash page cannot be re-programmed before it has been erased. However, since erase operations can only be performed on a block basis, overwriting a small page requires erasing a large block. This in return, requires copying out the valid data in the block, and writing them back after the block has been erased, leading to significant *write amplification*. To resolve such an issue, flash memory usually implements an out-of-place update mechanism, in which when a page is overwritten, it simply stores the new data to a new empty page, and marks the old page as invalid. The invalid pages will be cleaned and the space will be reclaimed later by *garbage collection* (GC). Figure 1 shows a concrete example: For HDDs, when an over-write happens, the same physical storage is directly updated with the newly written data B. For flash memory however, to handle an overwrite, it places the newly written data B on a new physical page, and marks the page that previously stores A as invalid. GC is periodically performed to reclaim space occupied by invalid pages following these steps [42]: (1) select those blocks which satisfy certain reclaim criteria (e.g., the number of invalid pages exceeds a threshold) as victim blocks; (2) copy the valid data stored in the victim blocks to free blocks; (3) erase the victim blocks.

Second, each flash block only has a limited number (e.g., 10K) of program-erase (P/E) cycles before it is worn out and cannot reliably store information. To prolong the service life of flash memory, *wear leveling* is usually required, by which writes/erasures on flash memory are distributed evenly across the entire flash such that no single block will have significantly larger P/E cycles than others.

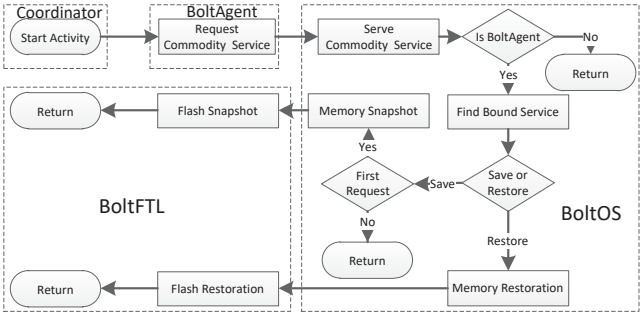


Figure 3: `Bolt` work-flow.

Flash translation layer. To be compatible with block-based file systems (e.g., EXT4, FAT32), a flash storage medium is usually used by emulating it as a block device (we call it a *flash-based block device*). This can be achieved by introducing a special *Flash Translation Layer (FTL)*, which transparently manages the special nature of raw flash and provides a block-based access interface. As it is shown in Figure 2, FTL translates the logical page addresses (LPA) from the upper layer (e.g., file systems) to the physical page addresses (PPA) of the underlying raw flash. This requires a data structure which can maintain the mappings between LPAs and PPAs.

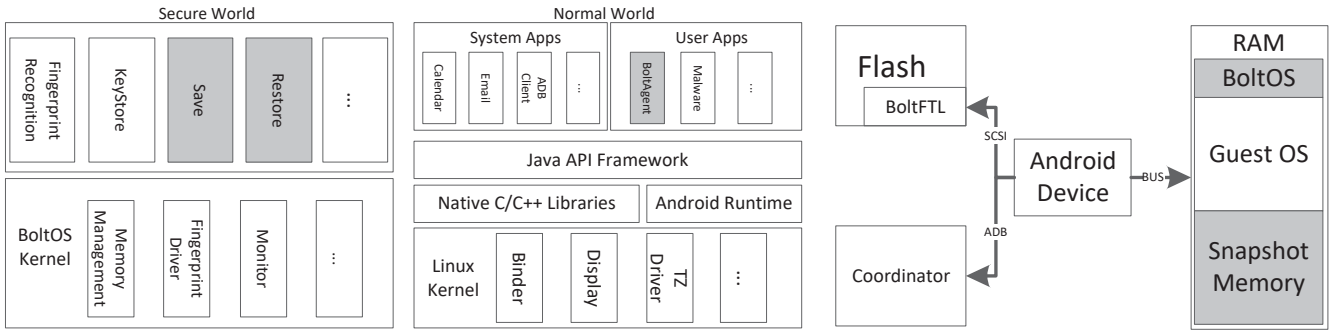
3 DESIGN

This section presents the design of `Bolt`. We start with describing our threat model and assumptions. We then give an overview of `Bolt`. Finally, we elaborate the design details of memory recovery and flash recovery in `Bolt`, respectively.

3.1 Threat Model and Assumptions

In `Bolt`, we consider aggressive malware which can obtain ultimate privilege in the normal-world Android system. That is, it can execute arbitrary code in all the privilege levels of the system, including kernel. This can be achieved by exploiting kernel vulnerabilities to escalate privileges. With kernel-level privilege, we assume the malware could further break protections enforced by SELinux policies. In this way, it could obtain write permission to system partitions.

We assume that the guest system relies on TrustZone-based TEE for security-critical services. In practice, almost all the smartphones available in the market today are equipped with TrustZone, and newer releases of Android have even standardized the interface between the TEE and the normal-world Android OS [18]. `Bolt` relies on existing code in Android to invoke the snapshot services in the secure world. Therefore, `Bolt` does not require any modifications into the Android system. In addition, we assume that the light-weight OS in the secure world, i.e., `BoltOS`, is resilient to attacks from normal world. This is a widely acceptable assumption in the domain of TrustZone-based security solutions, although we admit that there are some real-world attacks that could compromise the TrustZone secure world [27].



(a) Overview of the normal-world Android OS and secure-world `BoltOS`. Shaded components are newly added. As shown in the Figure, the entire Android system is unmodified except for a trivial non-suspicious user space app, `BoltAgent`.

(b) Architecture overview. Shaded memory regions are secure resources that cannot be accessed by the normal-world Android System.

Figure 4: System overview

3.2 Overview

`Bolt` comprises of four main components: `BoltAgent`, `BoltOS`, `BoltFTL`, and `Coordinator` (see Figure 3).

First, an in-guest app called `BoltAgent` runs in the Android system (Figure 4a). This app is not suspicious in the sense that it does nothing other than invoking two different requests of commodity security services that are already implemented in the TEE of the Android system, such as fingerprint recognition or trusted keyStore. `BoltOS` in the secure world receives the requests from `BoltAgent` as cues for save/restore operations.

Second, in the secure world, we run `BoltOS`, a lightweight OS that is responsible for handling commodity security service requests and add-on restoration services. In particular, a commodity security service, such as fingerprint recognition, is bound with the `save` service, and another service, such as keyStore, is bound with the `restore` service. As shown in Figure 3, when `BoltOS` receives a request from `BoltAgent`, apart from serving the ordinary commodity request, `BoltOS` executes the corresponding add-on restoration service. A snapshot includes two parts – one for memory and the other for disk. `BoltOS` handles memory snapshot directly (Section 3.3), and forwards the disk snapshot request to the flash firmware, which we will describe in the next paragraph. We set aside a physical memory region of equal size with that used in the guest system (Figure 4b). To take a memory snapshot, `BoltOS` simply copies the entire physical memory assigned to the guest to the snapshot memory. In addition, the processor contexts, including the general purposed registers, *Current Processor State Register (CPSR)*, and *Translation Table Base Register (TTBR)*[3], etc. are saved in `BoltOS`. The restoration of physical memory is the reverse operation to snapshotting.

Third, `BoltFTL` is a piece of customized flash firmware. It receives customized SCSI commands [19] from `BoltOS` to perform `save` and `restore` operations to the flash. Taking advantage of the out-of-place update feature of flash, it is able to efficiently save and restore the content of the entire flash without time-consuming overwriting. Specifically, after receiving a `save` command, `BoltFTL` triggers an active garbage collection to store the clean-state data more compactly in flash blocks, and backs up the essential FTL metadata (e.g., the mapping table, which records the mappings between LPA and PPA) to a few reserved flash blocks. `BoltFTL` also regulates the write operations issued from the upper layer such that they will not overwrite those flash blocks that store the clean-state data. Garbage collection in `BoltFTL` is also revised such that the flash blocks storing the clean-state data will not be reclaimed. After having received a `restore` command, `BoltFTL` simply restores the backup metadata. Since the flash blocks storing the clean-state data are intact, the flash can be directly restored to the clean state. The details of `BoltFTL` will be elaborated in Section 3.4.

Finally, a `Coordinator` (Figure 4b) in a separate PC connects the guest device through the Android Debug Bridge (ADB) utility. It downloads the malware to the Android device, and invokes the `BoltAgent` to issue a `save` command to take a system snapshot after system boot and to issue a `restore` command whenever an analysis is completed. Note that these requests are issued from the normal-world Android system, which is subject to exploit. For example, the infected Android could issue a `save` command after infection. Therefore, the following `restore` commands would restore the system to the saved infected state. Taking another example, the infected Android system could simply refuse to route these commands to the secure world, leading to Denial of Service (DoS) attacks. We address the first issue by only accepting the `save` command for once and ignoring the others. Therefore, only the clean state after system boot can be saved. For the second issue, we set up a watch dog in the secure world. After a certain period of in-activeness, a `restore` command is forced to be executed.

3.3 Memory Recovery

This section describes how `BoltOS` makes memory snapshot of the running system. To begin with, we briefly introduce the architectural design of the ARM processors.

ARM architecture. Almost all the modern OSes work on virtual memory. ARM processors support virtual memory by a *Memory Management Unit (MMU)* and a set of auxiliary system configuration registers. Specifically, when the `M` bit of the *System Control*

[3]The TTBR register is the pointer to the first level page table, similar to CR3 in x86 processors.

Register (SCTLR) is set, the MMU is enabled. The following memory accesses will first go through the page tables that translate the virtual addresses to the actual physical addresses. The page table is a multi-level data structure, with the first-level base address pointed by the *Translation Table Base Registers (TTBR)*. To speedup address translations, ARM processors have a built-in *Translation Lookaside Buffer (TLB)* that caches the recently executed page translations. A TLB entry is indexed by the corresponding virtual address, plus a *Address Space Identifier (ASID)* that is uniquely assigned to individual tasks. Therefore, during task switches, the TLB does not need to be flushed.

In an ARM processor with TrustZone support, most of the system registers are banked, meaning that they have different copies in each world. This greatly simplifies the implementation of a standalone OS in the secure world. In particular, during world switches, the page table is automatically switched to the copy that is previously set in the destination execution environment, without the need to update TTBRs. In addition, the NS bit which indicates the current execution environment is also used to index a TLB entry, making TLB flushing unnecessary.

Snapshot and Restoration. Since BoltOS runs in an isolated execution domain, there is no circular dependencies between the physical memory and the processor context, which occur in in-guest restoration solutions [25].

The first step of making memory snapshot is to save the raw physical memory. In Bolt, we symmetrically partition the physical memory into two regions – one is loaded with the guest system, and the other is used to hold the snapshot of the guest system. The first region is a non-secure resource that can be access by both worlds, while the second one can only be access by BoltOS. Saving and restoring the physical memory is straightforward – BoltOS only needs to copy the guest physical memory to and from the snapshot region.

The physical memory is tightly coupled with the processor context. Changing physical memory without recovering the processor context will crash the system. For example, the new TTBR may point to a memory region that contains invalid entries in the snapshot image. As a result, the MMU immediate detects the unmapped virtual address and triggers a data abort. In BoltOS, apart from restoring the general purpose registers, we also recover the TTBR, SCTLR, and ASID registers. In addition, TLB is flushed to avoid recycling use of the same ASID at the time of saving and restoring.

3.4 Flash Recovery

Barely restoring the state of physical memory and processor is not enough, since the state of peripherals may be inconsistent with the system being recovered. Take flash as an example, after the memory is recovered, the kernel, which maintains data structures related to the file system, anticipates a matching back storage, which in fact has been infected. This inconsistency of critical data structures may cause system crash. More seriously, malware may retain a copy of itself in the non-volatile flash, which may be activated later to infect the recovered system. Therefore, it is necessary to save and restore the content stored in flash as well. Restoring flash by overwriting the entire content is time consuming, since flash is "update unfriendly" (Section 2.3). This will be exacerbated when the flash has a large capacity.

To enable fast recovery of flash, we take advantage of its out-of-place update feature. This special feature of flash ensures that during malware analysis, the malware cannot corrupt the clean-state data (specifically, the content) by over-writing them, which can be used to restore the clean state later. In addition, to avoid those data being damaged by garbage collection/wear leveling during malware analysis, we customize the flash firmware (FTL) by carefully modifying the existing garbage collection and wear leveling implementation in FTL. Note that modifying FTL is advantageous, since it stays between the OS and the raw flash, and is transparent to the OS. This allows our design to be resistant to malware that can obtain a kernel-level privilege.

The resulting design, BoltFTL, can support fast flash restoration after malware analysis. In the following, we will elaborate the main operations of BoltFTL. We mark off these operations into three phases. In the first phase, following the *save* command received from BoltOS, BoltFTL takes a snapshot of the clean-state flash. This phase is executed only for once because BoltOS only responds to the first *save* command. Then, in phase two, malware begins execution and infects the flash. Our customized BoltFTL ensures that malware can never damage pages storing clean-state data. In the final phase, BoltFTL recovers the flash to the saved snapshot directed by the *restore* command from BoltOS. After this, BoltFTL is ready to enter phase two.

Phase 1: Malware analysis in-preparation. In this phase, BoltFTL performs necessary operations to facilitate flash restoration. In particular, after receiving the customized *save* SCSI command, it performs the following steps to backup the clean-state data.

First, BoltFTL will trigger an active garbage collection such that user data can be stored in a compact manner. Specifically, BoltFTL marks the blocks having invalid pages as victim blocks, copies the valid data in these victim blocks to free blocks, updates the corresponding mappings, and finally, erases the victim blocks.

Second, BoltFTL makes a backup of essential metadata (such as the mapping table) to facilitate restoration. Note that the metadata is usually much smaller in size compared to the stored data. Like other reserved blocks (e.g., blocks reserved for wear leveling and bad block management), the blocks containing metedata backup are invisible to the upper layer. Phase one is execute only for once. Therefore, the clean-state metadata is backed up only for once. After metadata backup, the flash is ready to accept I/O requests issued by malware.

Phase 2: Malware analysis in-motion. During the malware analysis, BoltFTL carefully regulates flash operations to protect the integrity of the clean-state data in the reserved flash blocks:

- *Read.* A read operation does not affect flash integrity. Therefore, BoltFTL simply follows the same logic as that used in a conventional FTL.
- *Write.* As with a conventional FTL, BoltFTL adopts an out-of-place update mechanism to handle write operations. When allocating a free page, BoltFTL ensures that a page containing clean-state data or backup of metadata is *never* selected.

- *Garbage collection.* As mentioned earlier, BoltFTL also performs out-of-place updates, meaning that each write operation will be performed on a new flash page. Therefore, garbage collection is essential for the removal of stale data. To ensure that the clean-state data are stored intact, BoltFTL modifies garbage collection so that blocks storing clean-state data are *never* selected as victim blocks.
- *Wear leveling.* As BoltFTL does not reclaim the blocks storing clean-state data during garbage collection, the P/E cycles of these blocks would not increase over time. Eventually, uneven P/E cycles will appear among the blocks storing clean-state data and the others. To prolong the life of the flash, BoltFTL customizes wear leveling with the following logic that ensures an even P/E cycle distribution. (1) Whenever a free block is allocated for data writing, a wear leveling checking is performed. (2) If the P/E cycle of the block (denoted as $\mathbb{A}$) is higher than the average P/E cycle of all the blocks by a certain threshold, wear leveling is performed. (3) To perform wear leveling, BoltFTL selects the youngest block (i.e., the block with the least P/E cycles, which is denoted as $\mathbb{Y}$) as the new block for data writing. (4) If $\mathbb{Y}$ contains clean-state data, BoltFTL copies the clean-state data to the previously allocated block (i.e., $\mathbb{A}$), erases the young block $\mathbb{Y}$, and finally selects $\mathbb{Y}$ as the final block for data writing. Moreover, the mapping table and other metadata are updated accordingly.

Phase 3: Recovery from malware analysis. BoltOS sends a *restore* command to BoltFTL to notify the restoration of the flash. Upon receiving this command, BoltFTL simply discards the old metadata and activates the backup metadata by coping them to the RAM of the flash controller. In this way, the flash can be instantly restored to the clean state. Note that the reserved blocks storing backup metadata is never modified during phase two and phase three.

4 IMPLEMENTATION

We have implemented a proof-of-concept prototype for Bolt on an i.MX 6Quad SABRE experiment board, which integrates a four-core ARM Cortex-A9 processor, 1 GB DDR3 DRAM and 256 KB SoC internal RAM (iRAM). In the normal world, we run an Android 7.0 OS. Our prototype implements all the designed functions, except that we do not provide commodity security services in BoltOS. In fact, the trusted OS in the secure world is proprietary property in all the commercial products. In BoltOS, we only implement the save and restore services. In Android, the BoltAgent invokes a customized interface directly to kernel to request these services. We connect to our i.MX experiment board a programmable flash board LPC-H3131 [28] through USB interface. The flash board holds the System and Data partitions of the Android system. BoltFTL is built based on OpenNFM [8], an open source NAND flash controller framework. In the following, we detail the implementation of BoltOS and BoltFTL.

4.1 BoltOS

After the proprietary device ROM, BoltOS resumes execution in the secure world, and initializes itself within iRAM, which is a separated on-chip memory other than DRAM. In this way, we can symmetrically allocate the whole DRAM to the Android system and snapshot image. In particular, BoltOS configures the TZASC so that the first 512 MB of DRAM is set to be a non-secure resource for the Android system and the remaining 512 MB of DRAM is set to be a secure resource for snapshot storage. Finally, it loads the Android bootloader to the DRAM region assigned to the Android system, and switches to the normal world to run the Android bootloader, which further boots the Android OS.

The type of service requested by BoltAgent is indicated by the r0 register. BoltOS is linked with the newlib C library [46] for embedded systems. Hence, we can readily invoke standard C library functions such as memcpy to speed up memory saving and restoring. In total, BoltOS consumes less than 30 KB memory in iRAM, including 16 KB for page tables.

4.2 BoltFTL

We have implemented a prototype of BoltFTL using OpenNFM [8], an open source NAND flash controller framework. OpenNFM uses an architecture consisting of three layers. The highest layer mainly handles mappings between the LPA from upper layer and the PPA in raw flash, so that the flash-based storage device can provide a uniform block device interface to file systems. The middle layer mainly takes care of wear leveling and bad block management. The lowest layer provides a raw flash abstraction, handling the physical characteristics of different flash chips. We customized OpenNFM to work with LPC-H3131 [28], a development board equipped with 180 MHz ARM microcontroller, 512 MB NAND flash, and 32 MB SDRAM. The flash has 128 KB block size and 2KB page size, thus the entire NAND flash has 4,096 erase blocks, and each block is composed of 64 pages. Each mapping entry can be represented by 3 bytes, therefore the mapping table occupies 6 blocks.

In BoltFTL, to receive the save and restore commands from BoltOS, we take advantage of the reserved operation codes of SCSI commands [19]. Specially, we adopt an SCSI command with operation code *0x61H* to inform BoltFTL to start performing save operations, and adopt an SCSI command with operation code *0x62H* to inform BoltFTL to start performing restore operations.

5 EVALUATION

In this section, we evaluate the proposed system. First, we measured the time required to restore the guest Android system. We also break down the whole process to discover the most time consuming stage. Since the restoration time is highly dependent on the configuration of a real hardware, we also measured the time spent with different configurations of hardware. Following this, we dissected the time spent on flash restoration. Finally, we measured the runtime performance of flash access with our modified firmware. If the performance is severely influenced, the malware may observe such environmental change and refuse to expose malicious behaviors.

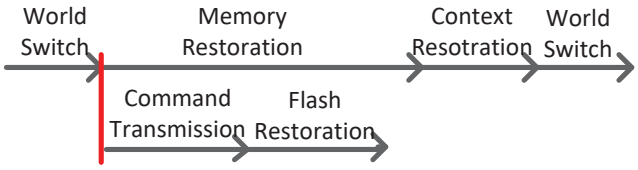


Figure 5: Restoration timeline

Table 1: Time breakdown for restoring the guest system (in μs).

World Switch	Memory Restoration	Context Restoration	Flash Restoration
1.2	2798087	23	433917

Table 2: Memory restoration under different physical memory sizes.

Memory Size (in MB)	384	448	512
Time (in μs)	2092653	2445271	2798087

5.1 Restoration Performance

We report the device restoring time in this section. As our experiment board has only 1 GB DRAM, we measured the time required to restore an Android system with 512 MB physical memory. The flash has 512 MB storage, and BoltFTL is able to recover the entire contents at a time. All the experiments ware performed for at least 10 times, and the averaged time is reported.

Breakdown measurements. Figure 5 shows the timeline of a restoration process. After receiving the restoration command, BoltOS issues the customized SISC command to BoltFTL. This process is non-blocking, therefore, BoltOS immediately begins to restore the physical memory and processor context. Finally, BoltOS transfers control to the guest system to resume execution at the time snapshot was taken. Note that in our experiments, the time required to restore the physical memory is far longer than that required to restore the flash. Hence, only the upper line in Figure 5 accounts for the time spent on system restoration.

To measure the time spent on each stage, we utilized the *Performance Monitor Unit (PMU)* available in our experiment board, and counted the CPU cycles spent. The time required was calculated as the elapsed cycles divided by the processor frequency. We temporarily disabled the Linux perf support in the normal world to eliminate its interference with the PMU state. We cannot measure the exact time spent on command transmission, because the clocks in BoltOS and BoltFTL are not synchronized. Rather, we recorded in BoltOS the time spent on command transmission and flash restoration as a whole. In Table 1, we list a breakdown measurement of time spent on each stage. As shown in the table, most of time was spent on the memory restoration. Based on the timeline shown in Figure 5, the total time to recover the system is 2.80s.

Restoration under varying memory sizes. We also assigned different sizes of physical memory to the guest system. Restricted by the size of DRAM on the device, the maximum tested size is 512 MB. In addition, Android system has minimal requirement for physical memory. Therefore, we only tested memory sizes between 384 MB and 512 MB. As shown in Table 2, the time consumption basically follows a linear relationship with the restored memory size. Note that the time for memory restoration depends on not only the physical memory size, but also the underlying SoC. A more advanced chip with a powerful processor and high memory bandwidth could further reduce the restoration time.

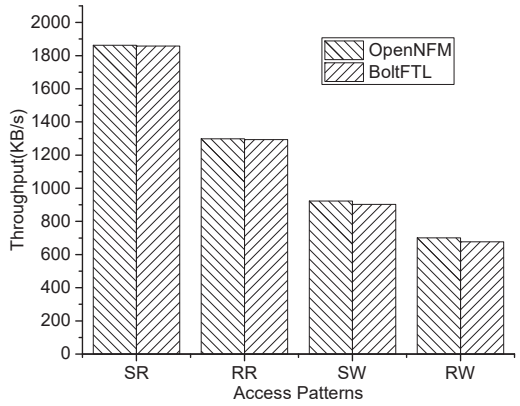


Figure 6: Throughput comparison between OpenNFM and BoltFTL. SR - sequential read, RR - random read, SW - sequential write, RW - random write

Evaluating the efficiency of flash restoration. To restore a flash-based block device, BoltFTL simply activates the backup metadata to replace the current one in the RAM (equipped with the flash-based block device). In our evaluation with LPC-H3131, this takes approximately 0.43s.

5.2 Flash Runtime Performance

BoltFTL is a customized flash firmware incorporating specifically designed strategies that feature snapshot function for the whole chip. To figure out how these strategies affect the performance of the flash, we used the AndroBench storage benchmark to measure the performance of the default OpenNFM and BoltFTL. We set the buffer sizes for sequential and random accesses to 32768 KB and 4 KB respectively (default values). As shown in Figure 6, we have the following observations.

(a) The throughput of read performances measured in BoltFTL is almost the same as the default OpenNFM. Indeed, read operations do not make any modification to the flash state, so the logic to process read operations is the same in BoltFTL and OpenNFM.

(b) The throughput of write operations measured in BoltFTL is slightly lower (2%) than the default OpenNFM. The introduced overhead is caused by the following reasons. (1) To increase the write throughout, FTL prepares a certain number of free blocks for upcoming writing operations in advance. This means these blocks need to be reclaimed by GC beforehand. However, when running with BoltFTL, the firmware does not select the blocks storing the clean-state data as victim blocks in GC, which inevitably increases GC frequency slightly to satisfy the demand of free blocks. (2)

GC in BoltFTL introduces much more uneven P/E cycles among flash blocks than default OpenNFM. To prolong the lifetime of the flash-based block device, in BoltFTL, wear leveling is invoked more frequently to ensure that programmings/erasures distribute evenly across the entire flash.

6 DISCUSSION AND FUTURE WORK

Decreased flash capacity. To protect the blocks that hold the clean-state data, BoltFTL never reclaims these blocks for reuse. Therefore, operations that need new block allocations will be influenced. For example, if the malware deletes a file that exists in the snapshot image, the number of available logical blocks is increased accordingly, whereas the number of total reclaimable physical blocks is not. As a result, the flash capacity is decreased. Fortunately, a clean Android system partition only occupies 268 MB in our build, indicating that there is at most 268 MB of storage loss. This will not be a problem for modern commercial flash-based block devices with high storage capacities.

Ware-and-tear artifacts. Bolt assumes that the malware could not distinguish it is being analyzed by running it in a bare-metal hardware without modification to the guest. However, as revealed by Miramirkhani et al. [29], more sophisticated evasive malware is able to exploit the "wear and tear" artifacts that inevitably occur on devices of real users, but not in-lab devices, to identify it is being analyzed. The authors also developed a statistical model to aid building system images that exhibit a realistic "wear and tear" state. Note that their findings do not indicate bare-metal analysis can be largely detected. Rather, existing solutions, including Bolt, can improve the fidelity by further taking the "wear and tear" artifacts into consideration.

Self restoration. In the current prototype, BoltOS in the secure world is only responsible for taking snapshot and making restoration for the normal-world guest system as required. However, in a real-world setting, BoltOS should also provide security services for the commodity guest OS. To exhibit a consistent view for the guest OS after restoration, the statuses of security services in BoltOS should be restored as well. Chicken-and-egg problem occurs here because there is no lower level execution domain than the TrustZone secure world to restore the status of BoltOS. In practice, this inconsistence would cause exceptions to apps in the guest system. For example, the security service typically returns a fault code to the guest if it cannot find a session associated with the guest-provided session id. Fortunately, the security services can be recovered after re-establishing the session. Moreover, for BoltOS, we do not need to restore the entire OS status as is done for the guest OS. Application level restoration, which has been well studied [34], could be employed for the purpose of security service restoration.

Selective restoration. Our prototype takes memory snapshot by simplify saving the entire physical memory assigned to the guest. However, there exists a lot of unused pages that do not need to be restored. As the physical memory becomes large, more time is wasted on copying unimportant pages. In the future, we plan to employ VM introspection techniques [22] to analyze the structure of physical memory, and only take snapshot of the pages in use.

7 RELATED WORK

This section reviews approaches used in malware analysis on both bare-metal and emulated platforms. We focus on the bare-mental setting, as it leaves minimal artifacts for the malware. We describe two fundamental challenges and their solutions in this setting – system restoration and malware behavior extraction. Finally, we introduce several works that take advantage of ARM TrustZone or flash for security purposes.

7.1 Malware Analysis on Bare Metal

System restoration. BareDroid [30] provides a quick restore mechanism that makes the bare-metal analysis of Android applications feasible at scale. However, it only restores the disk state of the system, so a reboot is needed to fully recover the system status. Bolt performs a complete system restoration including both memory and flash disk. BareBox [25] is a quick restoration system for bare-metal analysis on x86 machines. Both Bolt and BareBox are rebootless systems. In addition, they both enable the memory restoration by splitting the physical memory into two parts, and rely on a separate OS to take snapshot and make restoration. Regarding disk restoration, BareBox relies on an overlay-based mirror disk, whereas Bolt relies on a customized flash firmware. The separated OS in BareBox runs with the same privilege as the guest system, so it can be easily detected by kernel-level malware, and even be disrupted. In Bolt, we employ ARM TrustZone to implement an isolated OS. In addition, existing works require in-guest components to assist restoration, whereas Bolt is completely transparent to the guest. This is made possible by a non-suspicious agent app that issues requests for security services that already exist in the system.

Behavior extraction and analysis. LO-PHI [40] leverages additional hardware sensors to monitor the disk operation and periodically poll memory snapshots. It achieves a higher transparency at the cost of incomplete view of system states. BareCloud [26] is an armored malware detection system; it executes malware on a bare-metal system and compares both disk-level and network-level activities of the malware with other emulation and virtualization-based analysis systems for evasive malware detection. The disk-level activity is extracted by comparing the effected disk status with the initial state. The network-level activity is captured on the wire directly. These works focus on malicious behavior extraction on bare-metal systems without installing any in-guest software components, while Bolt focuses on quick restoration after each malware infection.

Placing analytic code in the guest could significantly simplify the process of malicious behavior extraction. TaintDroid [12] is a system-wide information flow tracking tool. It provides variable-level, message-level, method-level, and file-level taint propagation by modifying the original Android framework. TaintART [44] extends the idea of TaintDroid on the most recent Android Java virtual machine Android Runtime (ART). VetDroid [55] reconstructs the malicious behavior of the malware based on permission usage, and it is applicable to taint analysis. DroidTrace [56] uses ptrace to monitor the dynamic loading code on both Java and native code level. Although these tools attempt to analyze the target on real-world devices to improve transparency, the modification to the

Android framework leaves some memory footprints or code signatures, and the ptrace-based approaches can be detected by simply check the `/proc/self/status` profile. Moreover, these systems are vulnerable to privileged malware.

Zhang et al. [53] propose MalT, a bare-metal debugging tool for malware analysis. Its core idea is to use System Manage Mode (SMM), a special CPU mode in x86 architecture, to increase the debugging transparency. Ninja [31], a follow-up system of MalT, provides a transparent malware analysis framework on ARM platform. Willems et al. [49] used branch tracing to record all the branches taken by a program execution. As pointed out in the paper, the data obtainable by branch tracing is rather coarse and this approach still suffers from a CPU register attack against branch tracing settings.

7.2 Malware Analysis via Sandboxing

DroidScope [52] rebuilds the semantic information of both the Android OS and the Dalvik virtual machine based on QEMU. CopperDroid [45] is a VMI-based analysis tool that automatically reconstructs the behavior of Android malware including inter-process communication (IPC) and remote procedure call interaction. DroidScibe [9] uses CopperDroid [45] to collect behavior profiles of Android malware, and automatically classifies them into different families. Since the emulator leaves footprints, these systems are naturally not transparent.

Ether [11] is a malware analysis framework based on hardware virtualization extensions (e.g., Intel VT). It runs outside of the guest operating systems by relying on underlying hardware features. BitBlaze [39] and Anubis [3] are QEMU-based malware analysis systems. They focus on understanding malware behaviors, instead of achieving better transparency. V2E [51] combines both hardware virtualization and software emulation. HyperDbg [14] uses the hardware virtualization that allows the late launching of VMX modes to install a virtual machine monitor and run the analysis code in the VMX root mode. SPIDER [10] uses Extended Page Tables to implement invisible breakpoints and hardware virtualization to hide its side effects. However, Ether, BitBlaze, Anubis, V2E, HyperDbg, and SPIDER all rely on easily detected emulation or virtualization technology [7, 35–37] and make the assumption that virtualization or emulation is transparent from guest-OSes.

More subtle, anti-anti-analysis technique for emulated environment has also been developed. It enjoys both the benefits of scalability in emulation-based analysis and high coverage in bare-mental analysis. In particular, with Droid-AntiRM [47], the analytic component is able to detect the condition statements that could trigger the malicious behaviors and rewrite them on the fly to force a malicious path. However, we regard it as a start of a new round of arms-race between the malware writers and detectors. A definitive solution could be bare-metal analysis that leaves no artifact at all.

7.3 TrustZone-based Isolation Systems

TrustZone provides an isolated execution domain apart from the commodity OS. Based on it, a lot of work has been proposed to meet various security requirements. TrustDump [43] builds an isolated environment to reliably dump the physical memory contents to a peripheral in case the guest OS is compromised or crashed. `BoltOS` augments TrustDump by also supporting memory restoration. Trusted Language Runtime (TLR) [38] and TrustShadow [16] are two systems that shield unmodified applications from a hostile OS. TZ-RKP [4] and Sprobes [15] monitor critical operations of an OS by routing privileged operations to the secure world for inspection. CaSE [54] extends TrustZone to execute self-contained applications inside the cache to defeat DRAM attacks.

7.4 Data Protection in Flash Memory

A few existing works explored data protection techniques in flash memory. DEFY [33] and DEFTL [21] investigated techniques which can hide sensitive data into the flash media. As another important direction of data protection, secure deletion ensures that sensitive data can be completely removed from storage media. NFPS [20] and TedFlash [6] explored novel techniques which can irrecoverably remove sensitive information stored in flash.

8 CONCLUSION

Dynamic analysis on bare-metal is a promising technique to reveal the malicious behaviors of evasive malware. This work focuses on a less studied, but important topic in bare-metal dynamic analysis, i.e., quickly restoring the guest system to a clean state without exposing any instrumentation indicators. The proposed solution, `Bolt`, takes advantage of two hardware features that are widely used in mobile platforms, to develop a reboot-less restoration solution without any modification to the guest system. With this spotless "sandbox", the state-of-the-art malware is not able to identify that it is being analyzed. This is particularly valuable for building a scalable bare-metal analysis platform with high throughput, especially considering the ever-growing number of mobile malware and the rapid evolution of envision techniques. Experimental results obtained from our prototype implementation show that, `Bolt` is able to restore a full system state in 2.80 seconds, outperforming all the existing solutions.

ACKNOWLEDGMENTS

The authors would also like to thank the anonymous referees for their valuable comments and helpful suggestions. The work is supported by the Army Research Office under Grant No.: W911NF-13-1-0421 (MURI), and the National Science Foundation under Grant No.: CNS-1422594, CNS-1505664, SBE-1422215, CNS-1422206, DGE-1565570, CNS-1718459, OAC-1738929. Jingqiang Lin and Luning Xia were partially supported by the National Natural Science Foundation of China under Grant No.: 61772518, 61602476.

REFERENCES

[1] Adamwallred. 2014. nvmtrace - A proof-of-concept automated baremetal malware analysis framework. (2014). https://github.com/adamwallred/nvmtrace.
[2] Tiago Alves and Don Felton. 2004. TrustZone: Integrated Hardware and Software Security. *White Paper* (2004).
[3] Anubis. 2009. Analyzing Unknown Binaries. http://anubis.iseclab.org. (2009).
[4] 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 *ACM CCS'14, 2014*. 90–102.
[5] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. 2012. Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies. In *Black Hat*.

[6] Bo Chen, Shijie Jia, Luning Xia, and Peng Liu. 2016. Sanitizing data is not enough!: towards sanitizing structural artifacts in flash media. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 496–507.

[7] Xu Chen, John Andersen, Z. Morley Mao, Michael Bailey, and Jose Nazario. 2008. Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In *Proceedings of the 38th Annual IEEE International Conference on Dependable Systems and Networks (DSN '08)*.

[8] Google Code. 2011. OpenNFM. https://code.google.com/p/opennfm/. (2011).

[9] Santanu Kumar Dash, Guillermo Suarez-Tangil, Salahuddin Khan, Kimberly Tam, Mansour Ahmadi, Johannes Kinder, and Lorenzo Cavallaro. 2016. DroidScribe: Classifying Android malware based on runtime behavior. *Mobile Security Technologies (MoST'16)* (2016).

[10] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. 2013. SPIDER: Stealthy Binary Program Instrumentation and Debugging Via Hardware Virtualization. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'13)*.

[11] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*.

[12] Enck, William and Gilbert, Peter and Cox, Landon P and Jung, Jaeyeon and McDaniel, Patrick and Sheth, Anmol N. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*.

[13] Nicolas Falliere. 2010. Windows Anti-Debug Reference. http://www.symantec.com/connect/articles/windows-anti-debug-reference. (2010).

[14] Aristide Fattori, Roberto Paleari, Lorenzo Martignoni, and Mattia Monga. 2010. Dynamic and Transparent Analysis of Commodity Production Systems. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*.

[15] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. 2014. Sprobes: Enforcing kernel code integrity on the trustzone architecture. *arXiv preprint arXiv:1410.7747*.

[16] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. 2017. TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '17)*.

[17] Claudio Guarnieri, Allessandro Tanasi, Jurriaan Bremer, and Mark Schloesser. 2012. The Cuckoo sandbox. (2012). https://www.cuckoosandbox.org/.

[18] Google inc. 2017. Trusty TEE. (2017). https://source.android.com/security/trusty/.

[19] INCITS. 2015. SCSI Command Operation Codes. (2015). http://www.t10.org/lists/op-num.htm.

[20] Shijie Jia, Luning Xia, Bo Chen, and Peng Liu. 2016. Nfps: Adding undetectable secure deletion to flash translation layer. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 305–315.

[21] Shijie Jia, Luning Xia, Bo Chen, and Peng Liu. 2017. DEFTL: Implementing Plausibly Deniable Encryption in Flash Translation Layer. In *Proceedings of the 24th ACM conference on Computer and communications security*. ACM.

[22] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. 2007. Stealthy Malware Detection Through Vmm-based "Out-of-the-box" Semantic View Reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, New York, NY, USA, 128–138.

[23] Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. 2014. Morpheus: Automatically Generating Heuristics to Detect Android Emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*. 216–225.

[24] Dhilung Kirat and Giovanni Vigna. 2015. MalGene: Automatic Extraction of Malware Analysis Evasion Signature. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. 769–780.

[25] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2011. BareBox: Efficient Malware Analysis on Bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC '11)*. 403–412.

[26] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2014. BareCloud: Bare-metal Analysis-based Evasive Malware Detection. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 287–301.

[27] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher 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 *Proceedings of the Network and Distributed System Security Symposium*.

[28] Mantech. 2017. LPC-H3131. http://www.mantech.co.za/. (2017).

[29] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. 2017. Spotless Sandboxes: Evading Malware Analysis Systems using Wear-and-Tear Artifactsy. In *2017 IEEE Symposium on Security and Privacy*.

[30] Simone Mutti, Yanick Fratantonio, Antonio Bianchi, Luca Invernizzi, Jacopo Corbetta, Dhilung Kirat, Christopher Kruegel, and Giovanni Vigna. 2015. BareDroid: Large-Scale Analysis of Android Apps on Real Devices. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC 2015)*. 71–80.

[31] Zhenyu Ning and Fengwei Zhang. 2017. Ninja: Towards Transparent Tracing and Debugging on ARM. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ning

[32] Jon Oberheide and Charlie Miller. 2012. Dissecting the android bouncer. *SummerCon2012, New York* (2012).

[33] Timothy M Peters, Mark A Gondree, and Zachary NJ Peterson. 2015. DEFY: A deniable, encrypted file system for log-structured storage. (2015).

[34] J. S. Plank, M. Beck, G. Kingsley, and K. Li. 1995. **Libckpt**: Transparent Checkpointing under Unix. In *Usenix Winter Technical Conference*. 213–223.

[35] D. Quist and V. Val Smith. 2006. Detecting the Presence of Virtual Machines Using the Local Data Table. http://www.offensivecomputing.net. (2006).

[36] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. 2007. Detecting System Emulators. In *Information Security*. Springer Berlin Heidelberg.

[37] Joanna Rutkowska. 2004. Red Pill... or how to detect VMM using (almost) one CPU instruction. http://www.ouah.org/Red_Pill.html. (2004).

[38] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2014. Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications. In *ASPLOS'14, 2014*. 67–80.

[39] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security*.

[40] Chad Spensky, Hongyi Hu, and Kevin Leach. 2016. LO-PHI: Low-Observable Physical Host Instrumentation for Malware Analysis. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*.

[41] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. 2013. Mobile-sandbox: Having a Deeper Look into Android Applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC '13)*. ACM, New York, NY, USA, 1808–1815.

[42] Raja Subramani, Haritima Swapnil, Niharika Thakur, Bharath Radhakrishnan, and Krishnamurthy Puttaiah. 2013. Garbage Collection Algorithms for NAND Flash Memory Devices–An Overview. In *Modelling Symposium (EMS), 2013 European*. IEEE, 81–86.

[43] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Sushil Jajodia. 2014. TrustDump: Reliable Memory Acquisition on Smartphones. In *in Proceedings of 19th European Symposium on Research in Computer Security*. 202–218.

[44] Mingshen Sun, Tao Wei, and John Lui. 2016. TaintART: a practical multi-level information-flow tracking system for Android RunTime. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*.

[45] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic reconstruction of Android malware behaviors. In *Proceedings of 22nd Network and Distributed System Security Symposium (NDSS'15)*.

[46] Vinschen, Corinna and Johnsto, Jeff . 1999. Red Hat newlib C Library. (1999). https://sourceware.org/newlib/.

[47] Xiaolei Wang, Sencun Zhu, and Yuexiang Yang. 2017. Droid-AntiRM: Taming Control Flow Anti-analysis to Support Automated Dynamic Analysis of Android Malware. In *Proceedings of the 33rd Annual Conference on Computer Security Applications (ACSAC '17)*.

[48] Lukas Weichselbaum, Matthias Neugschwandtner, Martina Lindorfer, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. 2014. Andrubis: Android malware under the magnifying glass. *Vienna University of Technology, Tech. Rep. TR-ISECLAB-0414-001* (2014).

[49] Carsten Willems, Ralf Hund, Andreas Fobian, Dennis Felsch, Thorsten Holz, and Amit Vasudevan. 2012. Down to the Bare Metal: Using Processor Features for Binary Analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'12)*.

[50] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. 2017. Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC.

[51] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. 2012. V2E: Combining Hardware Virtualization and Software Emulation for Transparent and Extensible Malware Analysis. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE'12)*. 12.

[52] Yan, Lok Kwong and Yin, Heng. 2012. Droidscope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security'12)*.

[53] Fengwei Zhang, Kevin Leach, Angelos Stavrou, Haining Wang, and Kun Sun. 2015. Using Hardware Features for Increased Debugging Transparency. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*.

[54] Ning Zhang, Kun Sun, Wenjing Lou, and Tom Hou. 2016. CaSE: Cache-Assisted Secure Execution on ARM Processors. In *The 37th IEEE Symposium on Security and Privacy (S&P)*. SAN JOSE, CA.

[55] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. 2013. Vetting undesirable behaviors in Android apps with permission use analysis. In *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*.

[56] Zheng, Min and Sun, Mingshen and Lui, John CS. 2014. DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability. In *2014 International Wireless Communications and Mobile Computing Conference*.