CacheKit: Evading Memory Introspection Using Cache Incoherence

Ning Zhang*, He Sun†‡, Kun Sun†, Wenjing Lou*, Y. Thomas Hou*
*School of Engineering, Virginia Tech, Blacksburg, VA
{ningzhang, wjlou, thou}@vt.edu
†Department of Computer Science, College of William and Mary, Williamsburg, VA
{ksun}@wm.edu
‡Institute of Information Engineering, Chinese Academy of Sciences, China
{sunhe}@iie.ac.cn

***Abstract*—With the growing importance of networked embedded devices in the upcoming Internet of Things, new attacks targeting embedded OSes are emerging. ARM processors, which power over 60% of embedded devices, introduce a hardware security extension called *TrustZone* to protect secure applications in an isolated secure world that cannot be manipulated by a compromised OS in the normal world. Leveraging TrustZone technology, a number of memory integrity checking schemes have been proposed in the secure world to introspect malicious memory modification of the normal world.**

In this paper, we first discover and verify an ARM TrustZone cache incoherence behavior, which results in the cache contents of the two worlds, secure and non-secure, potentially being different even when they are mapped to the same physical address. Furthermore, code in one TrustZone world cannot access the cache content in the other world. Based on this observation, we develop a new rootkit called CacheKit that hides in the cache of the normal world and is able to evade memory introspection from the secure world.

We implement a CacheKit prototype on Cortex-A8 processors after solving a number of challenges. First, we employ the Cache-as-RAM technique to ensure that the malicious code is only loaded into the CPU cache and not RAM. Thus, the secure world cannot detect the existence of the malicious code by examining the RAM. Second, we use the ARM processor's hardware support on cache settings to keep the malicious code persistent in the cache. Third, to evade introspection that flushes cache content back into RAM, we utilize physical addresses from the I/O address range that is not backed by any real I/O devices or RAM. The experimental results show that CacheKit can successfully evade memory introspection from the secure world and has small performance impacts on the rich OS. We discuss potential countermeasures to detect this type of rootkit attack.

1. Introduction

With the emergence of the digital age and the upcoming Internet of Things, embedded devices are playing an increasing role in cyber space. Network enabled printers, thermostats, and TVs can no longer be treated as isolated systems, despite their limited computation capability and power consumption. Many embedded devices serve as controllers of safety critical systems, such as human pacemakers, automobiles, and aircrafts. With the growing importance of these devices, the number of attacks targeting these less protected embedded devices is increasing [7], [33], [39].

ARM family processors are currently the most widely used processors, powering over 60% of all embedded devices [6], [23], including 4.5 billion mobile phones [12]. *TrustZone* is a hardware security extension offered by ARM to provide an isolated trusted execution environment [24]. This isolated execution environment is known as the *secure world*, while the non-secure execution environment is referred to as the *normal world*. Code executing in the secure world is isolated and protected from the untrusted rich OS in the normal world. A number of recent research efforts propose to use TrustZone to protect sensitive code and data in the secure world [53], [56], [65], [74], [77]. On the other hand, since the code in the secure world has the privilege to access the memory and CPU registers of the normal world, but not vice versa, system integrity checking and malware detection tools can be installed in the secure world to detect potential malware in the normal world [27], [68].

In this paper, we develop a new type of rootkit called *CacheKit* that can evade the TrustZone-based memory introspection mechanisms by taking advantage of TrustZone cache incoherence design. We observe and verify that code in one TrustZone world cannot access the cache content in the other world. In other words, even though the secure world has the privilege to access the memory of the normal world, it *cannot* access the cache contents of the normal world. In TrustZone, the processor cache between the normal world and the secure world is separated by an additional *non-secure* (NS) bit in the cache tags [15]. However, this flag in the cache line is not directly accessible by system software and none of the publicly available documents explicitly describe how this NS bit is controlled in the cache. After a systematic study of Cortex-A8 processors, we figure out that the cache lines are completely separated between the two worlds for the same physical memory location. It is an effective design for eliminating cache flush during world switches; however, a rootkit may exploit such cache incoherence to conceal its presence in the normal world.

A typical cyber attack on embedded devices consists of two main steps. The first step aims to gain root privilege by exploiting system vulnerabilities, and the second step is to establish stealthy and persistent control on the computing system by installing *rootkits*. Rootkit is a stealthy software that is designed to hide the existence of malicious logic in the system [3], [13], [14], [29], [37], [49], [63], [66].

Based on the discovery of cache-incoherence design of ARM processors, we design and implement a prototype of CacheKit on ARM Cortex-A8 processors. CacheKit loads and keeps malicious code in the normal world's cache and uses TrustZone's cache incoherence to evade introspections from the secure world. We solve three major challenges in the design of CacheKit.

First, CacheKit should load the malicious code only into the cache of the normal world, but not into random access memory (RAM). We adopt the *Cache-as-RAM* (CAR) technique to set up a memory space for rootkit execution, and we call this memory space *CacheKit Space*.

Second, CacheKit should always keep the malicious code and data persistent in the cache. We use the ARM hardware supports on cache setting to lock the cache lines of the malicious code so that it will not be evicted by the rich OS's cache replacement mechanism.

Third, CacheKit should remain stealthy and be able to evade both introspection from the secure world and detection from the normal world. Because the malicious code only exists in the processor cache but not in the RAM, CacheKit can evade introspection from the secure world. To evade detection tools in the normal world, we map the CacheKit space to unused I/O address space, which is not scanned by anti-virus tools in the normal world. Moreover, when the instrospector in the secure world attempts to extract the cache contents by flushing the cache, CacheKit can automatically erase the malicious data, since the mapped I/O memory in cache is not backed by any real I/O devices or RAM. CacheKit can defeat all DMA-based introspection, since the malicious code only resides in the cache.

We implement a CacheKit prototype that achieves all the design goals on cache loading, cache locking, and cache concealing on the i.MX53 development board [40] after first verifying the cache-incoherence design of the ARM Cortex-A8 processor, which is used by the development board. We quantify the performance degradation with different malicious code sizes. Then we propose several countermeasures to detect CacheKit with knowledge of its inner working details.

In summary, we make the following contributions.

1) We systematically study cache coherence in ARM TrustZone and discover a unique cache incoherence behavior that may be exploited to build a cache-based rootkit.
2) We design a new type of rootkit that can evade introspection from both the secure world and the normal world. We are able to maintain the incoherent state of the processor cache in the ARM TrustZone platform using the hardware-assisted cache locking mechanism along with a physical address manipulation technique.
3) We prove that such cache-based rootkits are real by implementing a prototype on ARM Cortex-A8 processors. The experimental results show that CacheKit has small performance impacts on the rich OS. We also present a number of countermeasures to detect and mitigate CacheKit.

The remainder of the paper is organized as follows. Section 2 describes some background on ARM TrustZone and ARM cache architecture. We present the CacheKit architecture in Section 3. A prototype implementation is detailed in Section 4. We evaluate CacheKit in Section 5 and present potential countermeasures against CacheKit in Section 6. We discuss the impacts of the new rootkit in Section 7. Section 8 discusses related works. Finally, we conclude the paper in Section 9.

2. Background

In this section, we provide background information about ARM TrustZone, ARM caching mechanism, and ARM physical address space organization, all of which are essential to our cache exploitation mechanism.

2.1. ARM TrustZone

TrustZone is a set of hardware security extensions supported since ARMv6. It consists of extensions on processor, memory, and peripherals to ensure complete system isolation for running secure code. The isolated environment in the TrustZone is often called *the secure world*, while the conventional operating environment is called *the normal world* or *the non-secure world*. The two worlds have different access

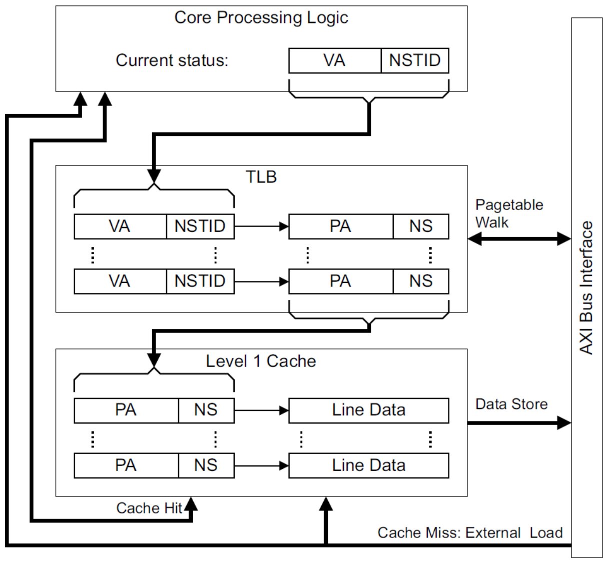

Figure 1. Caching in Trustzone [15]

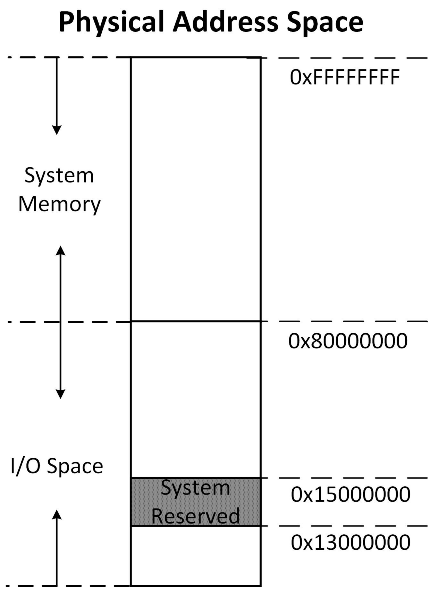

Figure 2. Physical Addressing on 32bit ARM System

privileges: The secure world can access most of the resources belonging to the normal world, yet the normal world cannot access any of the resources dedicated to the secure world. Within the security configuration register (SCR) of the *cp15* coprocessor, there is a *non-secure* (NS) bit that governs the security context of the processor. When the NS bit is cleared, the processor is in the secure world. When the NS bit is set, the processor is in the normal world. Memory and I/O devices are isolated by adding the new control signal (NS bit) to each read and write channel of the main system bus. All system resources are tagged with an NS bit. For memory, the addition of the NS bit to the bus transactions can be viewed as a 33rd address bit. There is a 32-bit physical address space for secure transactions and a 32-bit physical address space for normal transactions. For I/O transactions, the addition of the NS bit can be viewed as an access privilege token, and resource access will fail if the privilege is not correct.

2.2. ARM Cache

Cache is usually constructed with fast and expensive static random access memory. Modern processors often use *n-way set associative table* cache to enable parallel lookup operations. When TrustZone is introduced in the ARM architecture, the organization of processor cache is also modified. All levels of cache are extended with an additional tag bit, which records the security state of the transaction that accesses the memory [15]. Thus, in this cache coherence design, when the system switches between the two worlds, none of the cache lines need to be flushed. A secure cache line fill can evict a non-secure cache line, and vice versa [15]. This design significantly improves the performance of TrustZone. Cache is usually transparent to the OS, and it lacks fine-grained cache control with the exception of a small number of maintenance operations. However, to minimize cache pollution for certain embedded computing tasks, many ARM processors allow system designers to prevent cache lines from being evicted by locking them down.

Figure 1 shows the caching mechanism in TrustZone. The virtual address (VA) to physical address (PA) translation is put into the Translation Lookaside Buffers (TLBs), associated with a Non-secure Table IDentifier (NSTID) that permits secure and non-secure entries to coexist. The TLBs are enabled in each world from a single bit in CP15 Control Register. The level 1 cache stores the memory data at the PA, where an NS bit marks if the cache line belongs to the secure world or the normal world. This NS bit is set by hardware and it is not directly accessible by system software. In most modern processors, cache is physically indexed, physically tagged (PIPT). When the cache lines are PIPT, cache contents correspond only to the physical address. This enables us to tie a physical address range to a line in the cache.

2.3. ARM Physical Address Space

The entire range of memory addresses accessible by the processors is often referred to as *physical address space*. The length of such address space usually is not equal to the amount of actual physical memory installed on the platform.

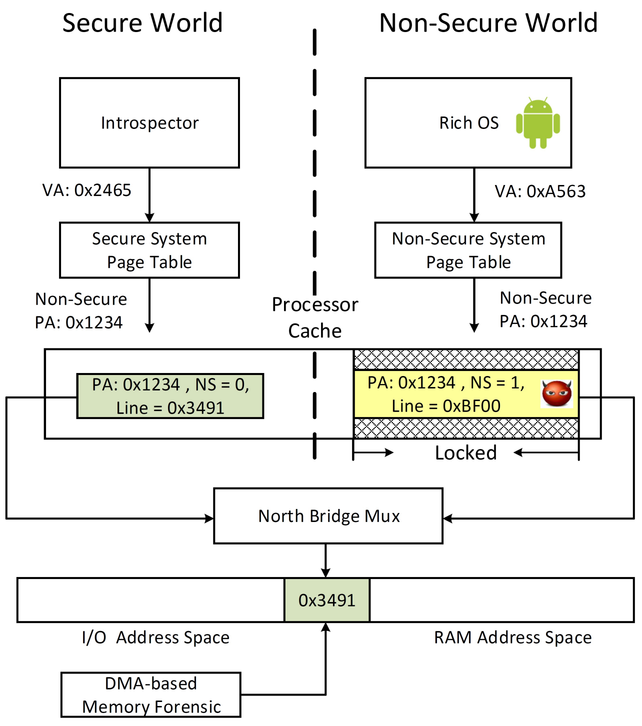

Figure 3. CacheKit Architecture

This is because some of the address ranges are mapped to the bus for I/O devices instead of dynamic random access memory (DRAM). A typical memory layout of a 32 bit ARM system is shown in Figure 2. The address range from 0x0 to 0x7FFFFFFF is backed by I/O devices, and the range from 0x80000000 to 0xFFFFFFFF is backed by system memory.

This memory layout is used by the Memory Map Unit (MMU) to route memory requests from the processor to either DRAM or memory-mapped I/O (MMIO). Even though the I/O space spans 2GB of address space, some of the area is actually left unused. For example, on i.MX53, address space from 0x1300000 to 0x1500000 is marked as unused system reserved.

3. CacheKit Architecture

CacheKit is a stealthy cache-based rootkit that can evade introspection from both the secure world and the normal world on ARM TrustZone architecture. The overall architecture of CacheKit is shown in Figure 3. The commodity rich OS resides in the non-secure world, while the introspector that performs the integrity checking runs in the secure world.

Both the rich OS and the introspector have their own page tables, called *non-secure system page table* and *secure system page table*, respectively, to translate virtual addresses to physical addresses. Even if two virtual addresses are different in two worlds, they may be translated into the same physical address. Intuitively, the stored value at the physical address should be same for both secure and non-secure world; however, the cache entry may be different in the two worlds even if the physical address points to the same location in the RAM. CacheKit exploits this cache incoherence design between the two worlds in TrustZone to conceal malicious code and data from the introspector running in the secure world. Since the malicious code resides only in the cache of normal world, CacheKit can escape from the detection of both the introspector and the DMA-based memory forensics hardware, which can only access the content in DRAM. This cache incoherence can be maintained with hardware cache locks in ARM processors. We can further improve the stealthiness of CacheKit by mapping to unused I/O address space in order to defeat detection by antivirus tools in the rich OS.

3.1. Cache Incoherence in TrustZone

All TrustZone enabled processors augment their cache with an NS bit so that the cache controller is able to distinguish if a cache line is a secure cache or a normal cache [15]. Since the NS bit in cache tag is governed by the security context of the processor, even though the secure world can access the RAM of the normal world, it cannot access the cache lines for the normal world. As shown in Figure 4, cache contents at the same physical address can be different for the two worlds. Therefore, if the malicious code can be saved in the normal world's cache, memory forensic tools in the secure world would not be able detect it.

Since no details have been provided in public documents on how various settings affect the cache behavior between the secure world and the normal world, we perform a systematic study on cache behaviors in ARM TrustZone and verify the identified cache incoherence problem on our prototype platform. For example, through real experiments, we observe that the effects of a cache flush depend on the world in which it is performed. A cache flush in the secure world will flush all the cache lines, regardless of the NS bit of the cache line. However, when the cache flush is performed in the normal world, it will only flush the normal world cache, namely, the cache lines with NS = 1.

3.2. Cache Exploitation

To exploit this cache incoherence issue in TrustZone, we need to tackle the challenges of loading the code only into the normal world cache and maintaining an incoherent state in the cache. First, we adopt the *Cache-as-RAM* (CAR) technique to load the malicious code into the cache of the normal world, but not into RAM. Second, We use the ARM cache locking mechanism to achieve persistent existence, guaranteeing that the malicious code will not be evicted. Lastly, we apply the *physical address space manipulation* technique to further enhance the stealthiness of CacheKit against detection from both the normal world and the secure world. A typical cyber attack consists of two steps, gaining privileged access to the system and maintaining access to the system. We assume the malicious code has obtained the root privilege and focus on maintaining stealthy and persistent access to the system.

3.2.1. Cache Loading. The first step is to load the malicious code into the cache, but not into RAM. Cache memory is ubiquitous across ARM processor architectures and families, and the cache subsystem can be initialized with few instructions. We can use the CAR [55], [79] technique to achieve this goal. CAR was originally developed to allow system BIOS code to store the stack in the cache before the DRAM is initialized. CacheKit uses CAR to store malicious code in the cache exclusively.

The caching attributes on ARM platforms are controlled by a number of registers and the system paging table entry. We need to set a memory page as *Writeback* and *Readable and Writable*. Therefore, when memory locations are cached, reads come from cache line and may cause cache fills; writes update cache line but not the memory. Modified cache lines will be written back only when cache lines need to be deallocated or when cache coherence needs to be maintained. The paging table entry controls the caching strategy of the address location, which can be remapped via the Type EXtension (TEX) remapping capability in ARM. TEX remapping allows OS to have a finer granularity control over the page memory attributes. When the TEX remapping is enabled, memory attributes are now mapped to *Primary Region Remap Register* (PRRR) and *Normal Memory Remap Register* (NMRR).

3.2.2. Cache Locking. After using CAR to load the code into cache, we still need to keep it persistent inside the cache. Cache is designed to dynamically store a small subset of frequently used data or instructions with a fixed replacement policy. The processor cache is typically transparent to the system software. However, since a finer control of the cache is imperative in meeting run-time and energy constraints in some embedded systems, ARM processors (e.g., Cortex A8 of the ARMv7 family) offer a coarse-grained cache control that allows system software to lock certain cache ways. CacheKit makes use of this hardware-assisted cache locking ability to persistently conceal code in the locked cache.

Hardware-based cache locking is a well-known feature in multiple processor families, including ARM946 [17], CortexA8 [19], CortexA9 [16], and NViDIA Tegra [20]. However, it is not the only way to keep memory contents in the cache. When hardware support is absent, memory allocation can be crafted to eliminate certain cache evictions [50]. Though cache locking has been supported by hardware (e.g., i.MX53 in our prototype), we have to design and use it carefully since naive use of the cache lock would lead to the exposure of the rootkit. We will present the details in Section 4.

3.2.3. Cache Concealing. CacheKit must evade introspections from both the secure world and the normal world. Many modern rootkit analysis tools [9], [11] rely on accurate acquisition of system memory for online and offline analysis. There are two common methods to acquire physical memory in the system: extracting from the processor and extracting from a peripheral device using direct memory access (DMA). The DMA-based methods directly acquire the physical memory and cannot extract cache content in the processors. Due to the cache consistency design in TrustZone, CacheKit can evade introspection from the secure world by loading the code only into the normal world's cache.

In the normal world, when tools such as LiME [8] use a kernel module to read the physical memory from the processor, the memory acquisition result will reflect the cache values that contain the malicious code. To solve this problem, we use caching of an unused system I/O address range. Most I/O address ranges and their contents, such as memory mapped ROM and option ROM, are static and cannot be changed by the processor. Therefore, forensic examiners and rootkit scanners will not search this area at all. In fact, even if forensic examiners do want to scan these address spaces, it is difficult to determine which address is safe to read and which to skip since accidentally reading certain hardware control bits can halt and crash the system.

4. CacheKit Design and Implementation

CacheKit conceals malicious code and data by cultivating a false perception for the introspection tools. The idea is based on our observations of an undocumented cache incoherence design property between the secure world and the normal world. We design and implement a CacheKit prototype that satisfies all the requirements on cache loading, cache locking, and cache concealing on the i.MX53 development board.

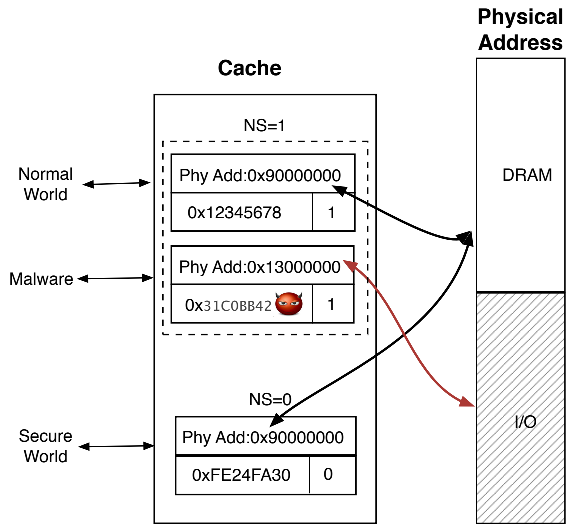

Figure 4. CacheKit Overall Design

The overall design of CacheKit is shown in Figure 4. The malicious code is loaded and locked in the normal world's cache lines, whose physical addresses point to a system reserved I/O address space. Since the secure world cannot access the cache content of the normal world, it cannot detect the rootkit that only resides in the normal world's cache. Furthermore, CacheKit utilizes processor cache entries mapped to physical addresses of a system reserved I/O address space which is not inspected by rootkit detection tools in the normal world. Design details are presented in the following.

4.1. Cache Incoherence Confirmation

On TrustZone enabled ARM platforms, the cache controller separates secure cache from non-secure cache using an NS bit in each cache line. A secure cache line fill may evict a non-secure cache line, and vice versa. When the system switches between the two worlds, there is no need to flush the cache lines. This design significantly improves the performance of TrustZone when switching between two worlds. However, through experiments on the i.MX53 development board, we confirm TrustZone's cache inconsistency property. In other words, each world cannot access the other world's cache content. In particular, the secure world cannot access the normal world's cache content, though it can access the normal world's RAM.

We design a series of experiments with the goal of exploring how various system settings affect the cache behavior between the two worlds. The first experiment set is to analyze how the modification of a non-secure memory area from the secure world gets propagated to the normal world. When the NS bit is set to 1 and cache is enabled in the secure world, changes made in the secure world are only visible to the secure world; the normal world can see the changes only after a cache flush happens in the secure world. We verify that when a modification is made in the secure world, the NS bit of the cache line is set to 0.

Secure Caching	NS Bit	Secure Rd	Normal Rd
disable	0	incoherent	coherent
enable	0	incoherent	coherent
disable	1	incoherent	coherent
enable	1	incoherent	coherent

TABLE 1. TRUSTZONE CACHE BEHAVIOR

The second experiment is to test our hypothesis that modification of memory in the normal world should only change the cache of the normal world. In other words, the change should not be visible to the secure world. Table 1 shows the experimental results that verify the correctness of our hypothesis. Secure Caching in the first column indicates whether the cache is enabled or disabled in the secure world. The NS bit is the non-secure bit in the security configuration register (SCR). It determines the current world of the processor. One exception is in monitor mode, where the processor is still in the secure world even when SCR.NS = 1. Secure Read and Normal Read are the results of a memory read after a memory write in the normal world, when the processor is in the secure world and the normal world, respectively. *Coherent* means the result of a memory read matches the value that we use for the memory write in the normal world, and *incoherent* means the result of memory read matches the value before the memory write.

The third set of experiments is to figure out how the NS bit in the cache is set in TrustZone. In other words, is the cache line's NS bit set by the processor execution mode, the SCR.NS bit, or physical memory address? If the NS bit in the cache line is set by the memory address, then there should be no observations of incoherent cache at all. This is because physical memory addresses are the same in both the secure world and the normal world for all the experiments. However, we can see from Table 1 that the NS bit of the cache line is not set by the memory address. Next, we test if the cache behavior is affected by the NS bit setting in the SCR register. The NS bit in the SCR determines if the processor is running in the normal world or in the secure world; however, when the processor is running in monitor mode, the execution is always in the secure world regardless of the NS bit. We compare all the secure modifications in monitor mode with the NS bit clear (i.e., the first two rows in

Table 1) to those with the NS bit set (i.e., the last two rows in Table 1). If the cache line NS bit is set according to the NS bit in the SCR, then these two sets of experiments should have different results. However, as Table 1 shows, the results are completely the same. Thus, we verify that the cache line NS bit is not set by the SCR.NS bit. Finally, we verify that the processor execution mode determines the cache value observed in each world regardless of whether secure caching is on or off. Therefore, since the cache lines are separated by the processor execution mode in TrustZone, cache lines in the normal world are only visible to the normal world and the secure world cannot access these lines.

4.2. Cache Incoherence Exploitation

Based on the cache incoherence issue in TrustZone, a stealthy cache-based rootkit can be successfully loaded and concealed in the normal world's cache through a three-step process involving cache loading, cache locking, and cache concealing.

4.2.1. Cache Loading. Processor cache is designed to be transparent to the system software, therefore there is no support in the ARM architecture to directly access the cache lines during normal operations. The only way to read/write a cache line is to have the processor read from or write to virtual memory. CacheKit's cache loading process consists of two steps.

The first step is to enable caching on the memory. In ARM, this is accomplished by setting the paging table memory attribute fields. Memory types in ARM can be either *Write-Back*, *Write-Through*, *Non-Cacheable* or *Strongly-ordered and Device*. All recent Linux kernels utilize the TEX remap feature in ARM, with which all memory attributes are coded using the PRRR register and the NMRR register. By writing both the TEX and B/C fields in the page table entry with codes from NMRR and PRRR, the memory page can be configured as write-back. With such configuration, LDR instructions on the page will trigger a cache line fill.

The second step in cache loading is to fill all the bytes of the code in cache. Due to the random replacement policy of the cache lines in i.MX53, it is necessary to make sure that the cache line fills are triggered only by the LDR or STR instructions used for filling in the cache. To avoid loading the memory of the program that is performing the cache loading, the code page has to be configured as Non-Cacheable.

4.2.2. Cache Locking. ARM processors offer coarse grain control of cache evictions. In particular, the Cortex A8 of the ARMv7 family allows system software to lock up to seven cache ways out of the total eight ways. As an example, the method to lock the contents at memory address *0x1234* in cache way 0 is as follows.

First, the cache corresponding to all the memory addresses to be locked in cache will need to be flushed out. This is to guarantee that a cache line fill happens in way 0. Memory contents can be filled in any one of the cache ways for a cache system that uses n-way associative table. If the cache line is already filled for *0x1234* in one of the eight ways, a LDR or STR instruction will no longer trigger a cache line fill. Therefore, without a cache flush, the cache line for memory address *0x1234* can be in any one of the eight ways.

Second, the L2 auxiliary cache control register is set to 0x000000FE so that only way 0 is unlocked and the other ways are locked. This is to make sure that for every line fill triggered by the LDR or STR instruction, it is allocated in the way we intend to lock at the end of the process.

Lastly, once all the program code is loaded as described in cache loading, then 0x00000001 is written to the L2 auxiliary cache control register to lock way 0 and unlock all other ways.

Note that even though cache locking is supported by the hardware, naive usage can still lead to exposure of the rootkit. This is due to the implementation dependent interaction between the locked cache line and the cache maintenance instruction.

The ability to maintain data in the volatile cache is crucial to CacheKit. On our ARM prototype platform, we use the cache locking capability supported by the hardware. For platforms that do not provide cache locking, it is still possible to preserve the state of the cache. For instance, cache line evictions for certain ranges of memory addresses can be eliminated with careful planning of memory allocation in the kernel [50].

4.2.3. Cache Hiding. There are two main problems with direct use cache locking. The first problem is introspection from the normal world kernel. Detection methods that use a kernel module to sequentially map each physical page into the kernel memory space [8] can still read the cache contents. The second problem is interaction of the locked cache lines with cache maintenance instructions. For instance, in ARMv7, several cache maintenance operations are available to the system software such as cache clean by set/way. Even when a cache line is locked, a clean operation can still cause the cache line to be written to memory. Thus, the rootkit can be detected once it is flushed out to memory.

To resolve the two problems with direct cache locking, we propose to carve a piece of usable memory space out of

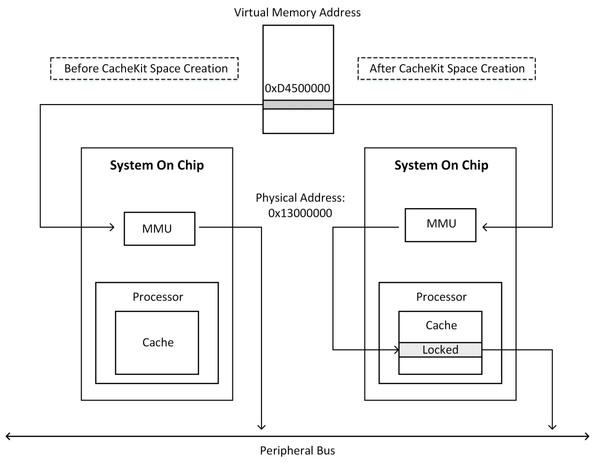

Figure 5. CacheKit on I/O Address Space

the physical I/O address space and then map it into cache. We call this newly created memory space the *CacheKit Space*. Typical memory allocation on the 32 bit ARM platform has the 0–2 GB range mapped to I/O space, and the 2–4 GB range is mapped to physical memory space. Figure 5 shows the CacheKit memory address mapping. Before deploying CacheKit, since address space between 0x13000000 to 0x15000000 belongs to the I/O range, any access requests to this area are redirected to peripheral bus by the MMU. However, after deploying CacheKit, since 0x1300000 to 0x15000000 is configured as memory space using the Cache-as-RAM technique, all read and write operations are redirected to the cache of the processor.

The CacheKit space is neither backed by any real RAM on the memory bus nor any real I/O device on the I/O bus. Therefore, when a transaction is sent to the bus, there is no physical device that will respond to the memory requests. To the best of our knowledge, none of the hardware on the commercial market respond to a failed cache flush to the bus. In other words, it does not record, report, or handle the error, and the invalid requests are simply ignored.

4.3. CacheKit Prototype

We implement a CacheKit prototype on the FreeScale i.MX53 mobile development platform, which features a single ARM Cortex A8 processor with 1GB DDR3 DRAM. The system boots with onboard flash along with the uboot and kernel supplied by the Micro-SD card. The image we used for our experiment is the FreeScale android 2.3.4 platform with a 2.6.33 Linux kernel. The ARM Cortex A8 processor has two levels of cache. L1 cache is a 4-way set associative cache with 128 set per way with 32 KB for instructions and 32KB for data. L2 cache is an unified 8-

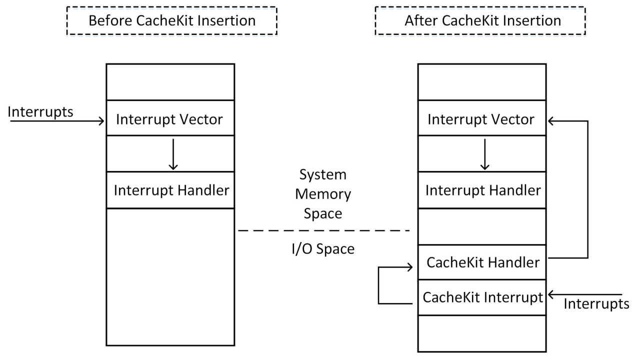

Figure 6. IVT CacheKit

way set associative table with 512 sets per way with a total size of 256 KB.

We first port an Android OS from secure domain to the normal domain based on the Board Support Package (BSP) published by Adeneo Embedded [2]. Furthermore, in order to perform the TrustZone cache experiments, a trusted boot loader as well as the monitor code must be loaded during the bootup process. The TrustZone initialization and monitor code are written in 500 lines of assembly and C code.

To implement a rootkit in the normal world, we choose to hook the interrupt vector table (IVT), similar to a previous work [34] that uses various hardware supports in ARM. IVT is the certainly not the only place to hook a rootkit; other choices include the system call table and specific event handling routines. We choose IVT for this prototype due to its simplicity. The address and handling of IVT has been relatively stable in the Linux kernel since version 2.6. The first step of rootkit insertion is to modify the address of the IVT to point to the shadow one in the CacheKit space. The page translation for the cache lines is then loaded and locked in the TLB. Once everything is set up in the CacheKit space, the page table in the kernel can be modified back to the original value. As shown in Figure 6, before the IVT is hooked, hardware interrupts go directly into the interrupt handling. After the insertion of CacheKit, all interrupts go through the CacheKit handler first.

Different types of payloads can be inserted into the CacheKit interrupt handler, such as network traffic sniffers, sms sniffers or encryption key sniffers. The payload in this prototype exfiltrates 1024 bytes of memory at a fixed memory location through the serial port DMA. CacheKit hooks interrupt 52 on the i.MX53 in order to make sure that the rootkit will be triggered whenever the home button is pushed. On a real world mobile deployment, the payload can sniff GPS location and exfiltrate through a network interface. Since GPS is not available on our experimental platform, we choose to dump memory through a serial port to demonstrate its capability.

We implement the rootkit as a kernel module with around 600 lines of C code. We write the payload in C code first, then compile it into approximately 360 lines of disassembly. The hex representation of the code is then converted into shell code, which is stored as data in the rootkit module and loaded directly into the cache. The CortexA8 processor can support locking of 1 to 7 cache ways with a maximum payload size of 224KB in the L2 cache. In order to minimize the use of L2 cache ways, we merge both the fake IVT and the CacheKit handler in one memory space in order to lock them into the same cache way.

5. CacheKit Evaluation

Our system evaluations consist of two main parts, one on the effectiveness of CacheKit and one on the impact of the rootkit on system performance.

5.1. Effectiveness of CacheKit

Three things must be verified to confirm the effectiveness of CacheKit. First, that the malicious code is indeed residing only in the cache and not in the DRAM. Second, that the malicious code can persist in the cache during normal system operation. Third, that CacheKit is able to evade rootkit detection from both the secure world and the normal world.

5.1.1. Cache Existence. One common method for checking if code or data resides in cache is to inspect the time taken to access the memory, since a cache-hit access is significantly faster than physical memory access. However, we cannot use this method since the presence of data in cache does not warrant its absence in physical memory. For instance, changes in cache can be flushed out to memory, so the data will exist in both cache and memory.

Instead, we use the *INVD* instruction provided by the processor to show the existence of data in cache and cache only. The INVD instruction is available in all of the major processor architectures, including ARM [18], [21], [22]. In ARM, the INVD instructions take two forms. The first form uses set number and way number to invalidate cache lines. The second form uses a modified virtual address, and the processor looks up the cache line associated with the modified virtual address and invalidates the cache line. The INVD instruction removes the cache content without processing it or forcing a write back to the physical memory. In other words, any changes on the memory addresses that are stored in the cache will be lost. By exploiting this unique feature, we can show that information is stored only in the cache if the memory contents match the previous value after the INVD instruction. We use the second form of INVD because the way allocation of a memory address is random. By invalidating the modified virtual address, we are able to observe the change of value at the memory address. Moreover, we can verify that the value does not change after we place our new information in the cache lines associated with the same memory address.

5.1.2. Cache Persistence. The volatility of the processor cache is a double edged sword for CacheKit. While it provides unprecedented stealthiness, the reliability of the rootkit is affected as well. Even though cache lines can be locked with hardware control, they still follow cache maintenance instructions. When a cache flush is invoked, the contents in the cache line do write out to the memory. However, since CacheKit space is mapped to an unused I/O address space, all cache contents will be lost once cache flush is invoked, since no backup memory exists. To verify that CacheKit can remain persistent in a real system, we use BBench [45] to continuously visit some popular websites, such as cnn.com, for six hours, then we check back to see if the data stored in the cache is still present. The experimental results show that the cache data persists. We also write a shell script to continuously run Linux commands for one night, and the cache data are still valid afterwards. On the system source code level, we search the kernel source code and find that there is only a single function in the kernel that would clean the entire cache, but it is never called in a uniprocessor deployment. With these experiments, we are confident that the cache locking mechanism on our platform is stable for stealthy rootkits.

5.1.3. Cache Elusiveness. We evaluate the cache elusiveness against TrustDump [68], a forensic memory acquisition toolkit based on TrustZone. This toolkit has a small piece of memory acquisition and integrity checking code stored in the secure world. We load CacheKit and perform two experiments. First, we use TrustDump to gather the addressable physical memory. Second, we use TrustDump to collect values from the I/O address space that Cachekit maps to. In both experiments, TrustDump cannot acquire contents in CacheKit Space. CacheKit can evade TrustDump in the first experiment because CacheKit space is not part of the physical memory in the physical address layout. CacheKit can evade the TrustDump in the second experiment due to the incoherent cache in TrustZone.

We also evaluate the CacheKit against rootkit detection in the normal world. First, we use LiME [8] to dump the physical memory of the experimental platform onto the SD Card. We then perform a binary pattern search on the

code size	baseline(0kb)	32kb	64kb	96kb	128kb	160kb	192kb	224kb
Linpack(s)	5.13	5.25	5.28	5.3	5.32	5.34	5.36	5.40
Linpack MT(s)	11.2	11.2	11.3	11.4	11.5	11.7	11.8	12.1
MemSpeed(s)	24.7	25.0	25.5	25.8	26.5	27.1	28.1	29.5
RanMem (s)	19	19.1	19.2	19.6	19.8	20.1	20.6	21.4

TABLE 2. CacheKit Benchmarks on Code Sizes

raw image to look for the rootkit signature. As expected, there is no trace of CacheKit found in the LiME extracted memory image. Thus, CacheKit can evade processor-based memory checking in the normal world. Second, we show that CacheKit cannot be detected by DMA-based rootkit detection. Since we do not have a dedicated hardware device with specialized firmware, we set up the serial port DMA to acquire physical memory. We conduct two experiments. The first one sets CacheKit in the legitimate memory area. The result shows that the acquired memory dump does not include the cache contents. The second test has CacheKit in the I/O address space at 0x13000000. We verify that the acquired data of the I/O address space only contains default data settings and not the cache contents.

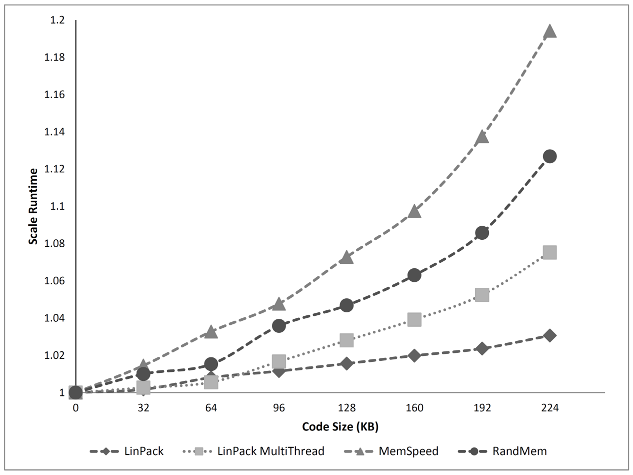

Figure 7. CacheKit Performance Impacts

5.2. Performance Impact

In CacheKit, all the malicious code and data reside in the cache. This implies that the larger the malicious payload, the more cache needs to be locked away. Since cache is designed to enhance system performance, the system may suffer from downgraded performance when the cache is locked intentionally for non-performance reasons. In this test, we are interested in quantifying the performance degradation with different levels of malicious code size. We test the system performance using benchmarks Linpack and Linpack MT [36], MemSpeed and RandMem [54]. Linpack is a software library for performing numerical linear algebra, and Linpack MT supports multiple threads. MemSpeed and RandMem are benchmarks for memory intensive operations. Table 2 shows the performance impacts with various code sizes.

The columns in Table 2 indicate the size of the malicious code. It is in 32 KB increments, because cache locking in i.MX53 operates on individual 32 KB cache ways of the L2 cache. Figure 7 and Table 2 show that system performance degrades as the size of the rootkit increases. The rootkits in these test are not hooked yet, therefore the performance degradation observed in these experiments are due to the locking down of cache lines to store the malicious code. The larger the rootkit, the more cache ways are locked. The Y axis shows the relative benchmark run time compared to the baseline in which none of the cache ways are locked. Even after code size is increased to 224 KB, Linpack only experiences 5% overhead for single thread computation and 8% for multi-thread computation, since the bottleneck for computation intensive tasks is not memory. However, for memory intensive operations, we observe a much bigger performance impact. For MemSpeed, which tests computation speed on a large area of memory, a maximum size rootkit will introduce a system performance overhead of 19.4%. For RandMem, which simulates random access to memory at various size, the maximum system performance overhead is 12%. Note that for many system workloads, the processor rarely uses all processor power to perform random memory access continuously. To get a sense of overall system impact, we also use the AnTuTu benchmark [5] to compare baseline with the maximal size rootkit and observe a 12% overall performance overhead.

6. Security Analysis

CacheKit is designed to conceal itself against the most advanced rootkit detection techniques. We demonstrate that it is possible for a rootkit in the normal world to evade introspection from a privileged process in the secure world by exploiting the unique cache design in ARM TrustZone.

Detection Methods	User	Kernel	VMBR [63]	SMMR [37]	Cloaker [34]	CacheKit
App level detection	✓	×	×	✓	×	×
OS level detection	✓	×	×	×	×	×
VMM level detection	✓	✓	✓	×	×	×
Coprocessor Based	✓	✓	✓	×	×	×
TEE Based	✓	✓	✓	×	×	×
Physical Memory Check	✓	✓	✓	✓	✓	×

TABLE 3. ROOTKIT TECHNIQUES AND DETECTION COMPARISON

One of the key design elements of CacheKit is to exploit the cache incoherency issue between the secure world and the normal world of TrustZone to evade introspection from the privileged secure world. Since switching between the two worlds does not flush the entire cache, CacheKit can remain persistent in the cache.

6.1. Evading Detection

A comparison between CacheKit and other types of rootkits against existing detection methods is provided in Table 3. Researchers propose to use the trusted execution environment (TEE) provided by hardware to perform the detection of rootkits in the memory [27], [57], [64], [68], [71]. All those solutions depend on the access capability of high-privileged TEE over the entire physical memory. With the malicious code hidden completely within the cache, there is no footprint in the physical memory, and it makes CacheKit an ideal technique for stealthy rootkit construction.

Detection at the application level usually scans the file system or firmware flash storage to match the checksum of files to a known good value or a rootkit signature. Since CacheKit does not modify the file system or firmware software, it cannot be detected by application level detectors. Kernel level rootkit detectors usually define a set of invariants in the kernel. Any alteration of these invariants [30], [72], [73], [78] is an indicator of rootkits. CacheKit is designed to only modify hardware configuration registers to hide the payload in the cache, so traditional kernel-based scanning cannot detect CacheKit.

Virtual Machine Monitor (VMM) based detectors that run in the hypervisor layer can access the entire physical memory. It is capable of detecting user level and kernel level rootkits. Detection of Virtual Machine Based Rootkit (VMBR), which exploits the same privilege layer, depends on the order of loading. If VMBR is loaded before the VMM detector, it is theoretically possible for the VMBR to create a nested virtualization environment to evade detection. However, since CacheKit resides in the cache and does not leave traces in memory, it can evade traditional hypervisor-based detectors. Moreover, when malware is aware of being executed in a virtualized environment [38], [41], it can simply stop all malicious activities. CacheKit can be augmented with this evasion logic to remain undetected.

Coprocessor-based detectors [61], [76] rely on a dedicated secure coprocessor in the system to interact with the system memory via peripheral bus. They access physical memory using DMA, which is independent of the processor. This physical level memory acquisition [32], [46], [58] simply obtains the memory content from the memory chip. Since CacheKit stores its entire code and data within the cache, all coprocessor-based approaches cannot find any malicious traces in the RAM.

Lastly, aside from the rootkit detectors mentioned above, it is also possible that certain system operations could affect the operation of the rootkit. System designers could perform a legitimate cache flush to maintain memory coherence in self-modified programs. Even though the interactions between locked cache lines and cache maintenance operations are implementation dependent, it is still possible that cache lines will be flushed out regardless of the lock flag. In fact, i.MX53 adopts this design. This problem is mitigated in CacheKit by redirecting cache data to I/O address space. When cache flushing happens, all data will be lost instead of being written back to memory and being detected.

6.2. Rootkit Paradox – Countermeasures

CacheKit provides a design paradigm to minimize the rootkit's footprint on the system by storing and running malicious logic in cache, leaving no trace in the memory at all. However, since it needs to stay accessible to the processor in order for the malicious code to be executed, the rootkit can still be detected when the defender knows exactly where to find it using the processor. This fundamental conflict between concealment and presence is known as *Rootkit Paradox* [51].

An examiner can detect CacheKit with the knowledge of its inner working details. Since the real hidden physical address of the IVT base address is hidden in TLB, the examiner can detect the IVT hooking by comparing the address translation performed by MMU and the address stored in the paging table. This value should remain constant after the system starts up, so it can be a cue that CacheKit or another rootkit is deployed. To remove CacheKit, the defender can first invalidate all TLBs, causing the system to use the real IVT. The cache can then be unlocked and flushed out. Thus, both the malicious mapping and malicious contents are removed with a one-time performance penalty. The order of the two operations is important. If the cache is first flushed, it may crash the system due to invalid interrupt handler addresses in the cleaned cache line.

Alternatively, the examiner can rely on a debugging interface such as JTAG to look inside the processor cache. When the cache locked register has bits set, the examiner should retrieve cache lines that are locked. Once a cache line is retrieved, the examiner can then check to see if the cache tag matches a valid memory range. If not, the contents of the lines should be further analyzed.

7. Discussion

7.1. CacheKit in Harden Environment

Despite the continuous efforts to eliminate vulnerabilities from the kernel, attacks that compromise the operating system remain a real threat [4]. The CacheKit loading process requires root privilege to modify sensitive registers and certain OS in-memory structures. The attacker can usually exploit one of the vulnerabilities in the kernel to obtain root privilege.

Recognizing such threats from rootkits, security researchers recently proposed to enforce kernel integrity using hardware support [27], [43]. When such kernel integrity protection mechanisms are present, it would be more difficult to deploy kernel rootkits, including CacheKit. The two fundamental building blocks in kernel integrity protection are the control of sensitive instructions in the kernel and the control of system page tables. When sensitive instructions are protected, there is no security sensitive instruction available to be directly called in the kernel code. Therefore even if the kernel control flow is hijacked, the security state cannot be changed without calling such instructions, and thus the damage is contained. When system page tables are protected, attackers cannot inject new instructions into the current code base. Thus, systems with these two properties can guarantee the integrity of the kernel code.

Modern kernels usually support extending its kernel code with device drivers or loadable kernel modules [27]. The new code to be inserted into the kernel has to be verified by a trusted entity, such as a security monitor in the secure world. With the cache incoherence problem we discovered, it is now possible for a hijacked rich OS to present to the monitor an incoherent view of the code to be inserted. The attacker can store the original image of the kernel module in RAM while placing security sensitive instructions in the cache lines. Since the monitor in the secure world cannot see the security sensitive instructions in cache, it will allow the installation of the kernel module. Now with the sensitive instructions in the kernel code, the hijacked rich OS can now redirect execution to exercise the newly inserted code and modify security sensitive system states, such as the location of IVT. We have to point out that due to the trap of the MCR instruction, a hijacked OS cannot use cache locking function in CacheKit. Thus, it may take many tries to load security sensitive instructions in the kernel. Lastly, even though it is theoretically straightforward, launching these attacks on a given system will require a significant amount of planning and a deep understanding of the targeted system architecture.

7.2. Rootkit Persistence

Rootkit persistence generally refers to the ability of a rootkit to survive power cycles. It typically requires modification of non-volatile storage; however, recent developments in disk forensics and integrity checking tools has forced attackers to adapt a memory-only approach [14], [29], [34], [63], [66], [75] in which non-persistent rootkits reside only in memory.

CacheKit is a new breed of non-persistent rootkit that brings a new level of stealthiness – it leaves no trace in either the non-volatile storage or the system RAM. However, this new level of stealthiness also brings drawbacks in its ability to remain persistent in the system. For example, cache contents are destroyed when the device powers off, and therefore CacheKit cannot persist after the system restarts. In the S3 sleep state of ARM processors [16], [19], the processor context is not retained, so Cachekit will have to have hooked into the power handling interface to save itself into SoC iRAM or I/O device memory to survive these power state changes. However, since people often keep mobile devices on for days without reboot, Cachekit poses a serious threat, similar to the well-known memory-only non-persistent rootkits [34], [63], [75].

Furthermore, with the current always-connected network architectures, attackers have other means to obtain persistence such as through watering hole attacks [44]. Lastly, it is possible to trade stealthiness for persistence if desired, such as storing of the logic in cipher text in memory to survive a power state change or infecting device drivers stored in the non-volatile storage for persistence over power cycles.

These changes will make CacheKit less stealthy but more persistent.

7.3. CacheKit Performance Impact

The use of cache as storage for malicious code and data has impacts on system performance. This limitation can be alleviated with careful planning of the cache placement. If the rootkit size is not as large as a single cache way, other heavily used kernel code can be locked in the cache way alongside the rootkit. This can reduce the performance impact caused by the locked cache way. On certain systems, appropriate use of cache locks can actually improve system performance [25].

Another closely related problem is the cache space optimization while applying CacheKit to existing rootkits. For example, the rootkit adore [14] modifies both the *task struct* and the *proc fs* function to hide processes and files. A simple way to conceal the two changes is to use one cache way for each modification. However, it is possible to fit two changes within one cache way, as long as the physical addresses do not multiplex to the same cache set. The tradeoff between system performance and the size of the malicious code needs to be carefully evaluated while designing a cache-based rootkit.

7.4. CacheKit on Other Platforms

One of the key enablers of CacheKit's evasion of TrustZone based detection is the cache coherency issue we observed in the experiments. The impacts of the techniques presented in CacheKit on mobile phones can be better assessed if such experiments can be performed on a series of different SoCs. Unfortunately, it proved to be very difficult to obtain TrustZone access on some of the most popular devices, such as the Google Nexus series and Samsung KNOX. However, since the coherency issue originates from the internal design of the processor, we believe the same problem applies to all platforms running Cortex A8. In the future, we would like to investigate this issue on different types of processors to obtain a better global picture.

CacheKit represents a design paradigm to hide malicious logic in the processor cache to evade detection. Some techniques used in CacheKit implementation are specific to the ARM architecture, such as the coarse-grained cache locking mechanism. And some of the implementations we designed on are hardware platform specific, such as the address of the invalid DRAM memory range. However, the general concept behind CacheKit, using cache as storage to evade memory forensic analysis, may be applied to other architectures. Since the control and internal organization of the processor cache is likely to be different for various processor architectures and families, rootkit developers need to have a deep understanding of the hardware platform to construct the CacheKit.

For instance, we are able to successfully map cache to reserved I/O address space on Intel Celeron processors. Although Celeron processors do not provide any cache locking mechanism, cache locking is available in Intel Xscale family processors [1]. In the future, we will extend our study on the cache behavior of other trusted execution environments, such as the new Intel SGX [31].

8. Related Work

Rootkits are a well-known category of malware. The primary goal of a rootkit is to conceal presence of attacker. It begins with simple file hijacking of binaries and libraries to cover up malicious activities. One of the early examples is the replacement of the *ls* command in Linux to hide files. Since these persistent rootkits need to modify non-volatile storage to survive system power cycles, file integrity checking tools [3], [49] can effectively detect them.

Later, malware authors developed in-memory rootkits that reside only in the operating system kernel memory [13], [14], [29], [66] to defeat the storage-based detection. To insert itself into the kernel control flow, in-memory rootkits either hook into legitimate kernel functions [10], [35], [52] or modify kernel data structures [13], [14], [29]. To detect this type of subversion, kernel level rootkit detectors first build a ground truth on a set of kernel invariants and then detect any alteration of these invariants [30], [72], [73], [78]. To enable offline rootkit analysis, researchers also propose to acquire the system memory using a dedicated secure coprocessor [61], [76] or physical hardware [32], [46], [58].

When both rootkits and their detectors have the same root privilege, it becomes a game of hide and seek between the attacker and the defender. Attackers move to obtain higher privilege than the kernel [37], [47], [63]. Virtual machine based rootkits (VMBR) [63] insert a customized malicious hypervisor beneath the currently running operating system. Firmware based rootkits infect the firmware on I/O devices [47] or the system BIOS [37]. However, rootkits are not alone in seeking higher privilege. New hardware and software rootkit detectors with higher privilege are also proposed [26], [27], [28], [42], [48], [57], [59], [60], [61], [62], [64], [71], [76].

Hypervisor is used in several research to perform introspection into the guest operating system [26], [42], [60]. However, since the virtualized environments can be reliably identified [41], malware can simply exit without execution. Moreover, new vulnerabilities are frequently found in the hypervisors [38]. Hardware features, such as security extensions in various processors (e.g., AMD SVM [21], Intel TXT [69], and ARM TrustZone [24]) have become a safe haven for defenders because of their ability to provide a trusted execution environment (TEE) with guaranteed isolation. Since the TEE has higher privilege, rootkit detectors can be installed in this environment [27], [57], [64], [68], [70], [71]. On the other hand, hardware features can also be exploited by rootkits to evade detection in the OS kernel [34], [67], [75]. For instance, Shadow Walker [67] exploits the *I-TLB* and *D-TLB* coherency problem in the Intel architecture to hide the rootkits. Cloaker [34] can hide the location of the replaced interrupt descriptor vector by locking the page translation in the translation look aside buffer.

In this paper, we propose a cache-based rootkit that keeps all malicious data and code in an incoherent TrustZone cache to evade rootkit detection. Besides TrustZone, CacheKit may work on other processors if they have a similar cache incoherence design.

9. Conclusion

In this paper, we present a systematic study of the cache incoherence behavior between the normal world and the secure world in the ARM TrustZone. Inspired by our observations, a rootkit called CacheKit is constructed to demonstrate the feasibility of concealing malicious code exclusively in the processor cache. CacheKit utilizes cache locking capability provided by the hardware along with physical address space manipulation to create an incoherent cache in unused I/O addresses. This incoherent state allows the rootkit to evade introspection from detection tools in TrustZone. Furthermore, the creation of new address space allows it to reside only in cache, making it untraceable through memory acquisition methods. Through this work, we hope to raise awareness of cache-based rootkit and foster advancement of defense mechanisms against rootkits.

Acknowledgments

This work is supported in part by US National Science Foundation under grants CNS-1217889, CNS-1446478, CNS-1405747 and CNS-1443889. Dr. Kun Sun is supported by U.S. Office of Naval Research under grants N00014-15-1-2396 and N00014-15-1-2012.

References

[1] 3rd Generation Intel Xscale Microarchitecture Developer's manual. http://www.intel.com/design/intelxscale/.

[2] Adeneo embedded. http://www.adeneo-embedded.com/.

[3] Advanced Intrusion Detection Environment. http://sourceforge.net/projects/aide/.

[4] Android rooting method : Motochopper. http://hexamob.com/how-to-root/motochopper-method/. Accessed: 2015-04-30.

[5] Antutu Benchmark. http://www.antutu.com/en/Ranking.shtml. Accessed: 2015-04-30.

[6] ARM holdings and Qualcomm: The winners in mobile. http://www.forbes.com/sites/darcytravlos/2013/02/28/arm-holdings-and-qualcomm-the-winners-in-mobile/.

[7] Hackers remotely kill a jeep on the highwaywith me in it. http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/.

[8] LiME, volatile memory acquisition. https://github.com/504ensicsLabs/LiME.

[9] Second look, forensic analysis tool. https://secondlookforensics.com/.

[10] Suterusu rootkit: Inline kernel function hooking on x86 and arm. https://github.com/mncoppola/suterusu.

[11] The volatility framework, malware and memory forensic. https://code.google.com/p/volatility/.

[12] World has about 6 billion cell phone subscribers, according to u.n. telecom agency report. http://www.huffingtonpost.com/2012/10/11/cell-phones-world-subscribers-six-billion_n_1957173.html.

[13] Infecting loadable kernel module. phrack, 2003.

[14] Stealth, Adore-ng rootkit. http://stealth.7530.org/rootkits/, 2003.

[15] ARM Security Technology, Building a Secure System using TrustZone Technology. apr 2009.

[16] ARM Cortex-A8 Processor Technical Reference Manual. may 2010.

[17] ARM946E-S (Rev1) System-on-Chip DSP enhanced processor - Product Overview. 2010.

[18] ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition. dec 2011.

[19] ARM Cortex-A9 Processor Technical Reference Manual. june 2012.

[20] Technical Reference Manual - VIDIA TEGRA 3. jan 2012.

[21] Advanced Micro Devices. Amd64 Architecture Programmer's Manual. Vol. 2, may 2013.

[22] Intel 64 and IA-32 Architectures Software Developer's Manual. sep 2013.

[23] ARM strategic report. http://ir.arm.com/phoenix.zhtml?c=197211&p=irol-reportsannual, 2014. Accessed: 2015-04-30.

[24] T. Alves and D. Felton. TrustZone: Integrated hardware and software security. *ARM white paper*, 3(4), 2004.

[25] K. Anand and R. Barua. Instruction cache locking inside a binary rewriter. In *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 185–194. ACM, 2009.

[26] A. M. Azab, P. Ning, E. C. Sezer, and X. Zhang. Hima: A hypervisor-based integrity measurement agent. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 461–470. IEEE, 2009.

[27] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 90–102. ACM, 2014.

[28] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 38–49. ACM, 2010.

[29] E. Balas. sebek, 2005.

[30] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 77–86. IEEE, 2008.

[31] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, 2014.

[32] E. Chan, S. Venkataraman, F. David, A. Chaugule, and R. Campbell. Forenscope: A framework for live forensics. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 307–316. ACM, 2010.

[33] A. Cui, M. Costello, and S. J. Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *NDSS*, 2013.

[34] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. Cloaker: Hardware supported rootkit concealment. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 296–310. IEEE, 2008.

[35] B. Dixon and S. Mishra. On rootkit and malware detection in smartphones. In *Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on*, pages 162–163. IEEE, 2010.

[36] J. Dongarra and P. Luszczek. Linpack benchmark. *Encyclopedia of Parallel Computing*, pages 1033–1036, 2011.

[37] S. Embleton, S. Sparks, and C. C. Zou. SMM rootkit: a new breed of OS independent malware. *Security and Communication Networks*, 6(12):1590–1605, 2013.

[38] P. Ferrie. Attacks on more virtual machine emulators.

[39] M. Fossi, G. Egan, K. Haley, E. Johnson, T. Mack, T. Adams, J. Blackbird, M. K. Low, D. Mazurek, D. McKinney, et al. Symantec internet security threat report trends for 2010. *Volume*, 16:20, 2011.

[40] Freescale. Imx53qsb: i.mx53 quick start board. http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=IMX53QSB&tid=vanIMXQUICKSTART.

[41] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is not transparency: VMM detection myths and realities. In *HotOS*, 2007.

[42] T. Garfinkel, M. Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, volume 3, pages 191–206, 2003.

[43] X. Ge, H. Vijayakumar, and T. Jaeger. Sprobes: Enforcing kernel code integrity on the TrustZone architecture. In *Proceedings of the 3rd IEEE Mobile Security Technologies Workshop (MoST)*. IEEE, 2014.

[44] W. Gragido. Lions at the Watering Hole: The VOHO Affair. *RSA blog*, 20, 2012.

[45] A. Gutierrez, R. G. Dreslinski, T. F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver. Full-system analysis and characterization of interactive smartphone applications. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 81–90. IEEE, 2011.

[46] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[47] J. Heasman. Implementing and detecting a PCI rootkit. 20(2007):3, 2006.

[48] X. Jiang, X. Wang, and D. Xu. 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*, pages 128–138. ACM, 2007.

[49] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 18–29. ACM, 1994.

[50] T. Kim, M. Peinado, and G. Mainar-Ruiz. System-level protection against cache-based side channel attacks in the cloud. In *USENIX Security*, 2012.

[51] J. D. Kornblum and C. ManTech. Exploiting the rootkit paradox with windows memory analysis. *International Journal of Digital Evidence*, 5(1):1–5, 2006.

[52] É. Lacombe, F. Raynal, and V. Nicomette. Rootkit modeling and experiments under linux. *Journal in Computer Virology*, 4(2):137–157, 2008.

[53] W. Li, H. Li, H. Chen, and Y. Xia. Adattester: Secure online mobile advertisement attestation using trustzone. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2015, Florence, Italy, May 19-22, 2015*, pages 75–88.

[54] R. Longbottom. Roy Longbottom's PC benchmark collection, 2014.

[55] Y. Lu, L.-T. Lo, G. R. Watson, and R. G. Minnich. CAR: Using Cache as RAM in LinuxBIOS. http://rere.qmqm.pl/~mirq/cache_as_ram_lb_09142006.pdf, 2006.

[56] C. Marforio, N. Karapanos, C. Soriente, K. Kostiainen, and S. Capkun. Smartphones as practical and secure location verication tokens for payments. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS'14, 2014.

[57] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 315–328. ACM, 2008.

[58] T. Müller and M. Spreitzenbarth. FROST. In *Applied Cryptography and Network Security*, pages 373–388. Springer, 2013.

[59] J. Oberheide, K. Veeraraghavan, E. Cooke, J. Flinn, and F. Jahanian. Virtualized in-cloud security services for mobile devices. In *Proceedings of the First Workshop on Virtualization in Mobile Computing*, pages 31–35. ACM, 2008.

[60] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 233–247. IEEE, 2008.

[61] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot-a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, pages 179–194. San Diego, USA, 2004.

[62] N. A. Quynh and Y. Takefuji. Towards a tamper-resistant kernel rootkit detector. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 276–283. ACM, 2007.

[63] J. Rutkowska. Subverting VistaTM kernel for fun and profit. *Black Hat Briefings*, 2006.

[64] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security Symposium*, volume 13, pages 223–238, 2004.

[65] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM TrustZone to build a trusted language runtime for mobile applications. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 67–80. ACM, 2014.

[66] sd. Linux on-the-fly kernel patching. phrack, 2002.

[67] S. Sparks and J. Butler. Shadow Walker: Raising the bar for rootkit detection. *Black Hat Japan*, pages 504–533, 2005.

[68] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia. TrustDump: Reliable memory acquisition on smartphones. In *Computer Security-ESORICS 2014*, pages 202–218. Springer, 2014.

[69] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[70] A. Vasudevan, J. McCune, J. Newsome, A. Perrig, and L. Van Doorn. CARMA: A hardware tamper-resistant isolated execution environment on commodity x86 platforms. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 48–49. ACM, 2012.

[71] J. Wang, A. Stavrou, and A. Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *Recent Advances in Intrusion Detection*, pages 158–177. Springer, 2010.

[72] Y.-M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting stealth software with strider ghostbuster. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 368–377. IEEE, 2005.

[73] Z. Wang, X. Jiang, W. Cui, and X. Wang. Countering persistent kernel rootkits through systematic hook discovery. In *Recent Advances in Intrusion Detection*, pages 21–38. Springer, 2008.

[74] J. Winter. Trusted computing building blocks for embedded linux-based ARM TrustZone platforms. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 21–30. ACM, 2008.

[75] N. Zhang, K. Sun, W. Lou, T. Hou, and J. Sushil. Now you see me: Hide and seek in physical address space. In *Proceedings of the 10th ACM symposium on Information, computer and communications security*. ACM, 2015.

[76] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer. Secure coprocessor-based intrusion detection. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 239–242. ACM, 2002.

[77] Y. Zhou, X. Wang, Y. Chen, and Z. Wang. ARMlock: Hardware-based fault isolation for ARM. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 558–569, 2014.

[78] F. Zhu and J. Wei. Static analysis based invariant detection for commodity operating systems. *Computers & Security*, 43:49–63, 2014.

[79] V. J. Zimmer, M. A. Rothman, and S. M. Datta. Using a processor cache as RAM during platform initialization, May 27 2004. US Patent 20,040,103,272.