CacheLight: A Lightweight Approach for Preventing Malicious Use of Cache Locking Mechanisms

by

Mauricio Gutierrez Barnett

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved May 2018 by the
Graduate Supervisory Committee:

Ziming Zhao, Chair
Adam Doupé
Yan Shoshitaishvili

ARIZONA STATE UNIVERSITY

August 2018

ABSTRACT

With the rise of the Internet of Things, embedded systems have become an integral part of life and can be found almost anywhere. Their prevalence and increased interconnectivity has made them a prime target for malicious attacks. Today, the vast majority of embedded devices are powered by ARM processors. To protect their processors from attacks, ARM introduced a hardware security extension known as TrustZone. It provides an isolated execution environment within the embedded device in which to deploy various memory integrity and malware detection tools.

Even though Secure World can monitor the Normal World, attackers can attempt to bypass the security measures to retain control of a compromised system. CacheKit is a new type of rootkit that exploits such a vulnerability in the ARM architecture to hide in Normal World cache from memory introspection tools running in Secure World by exploiting cache locking mechanisms. If left unchecked, ARM processors that provide hardware assisted cache locking for performance and time-critical applications in real-time and embedded systems would be completely vulnerable to this undetectable and untraceable attack. Therefore, a new approach is needed to ensure the correct use of such mechanisms and prevent malicious code from being hidden in the cache.

CacheLight is a lightweight approach that leverages the TrustZone and Virtualization extensions of the ARM architecture to allow the system to continue to securely provide these hardware facilities to users while preventing attackers from exploiting them. CacheLight restricts the ability to lock the cache to the Secure World of the processor such that the Normal World can still request certain memory to be locked into the cache by the secure operating system (OS) through a Secure Monitor Call (SMC). This grants the secure OS the power to verify and validate the information that will be locked in the requested cache way thereby ensuring that any data that remains in the cache will not be inconsistent with what exists in main memory for inspection. Malicious attempts to hide data can be prevented and recovered for analysis while legitimate requests can still generate valid entries in the cache.

A special thank you to my supervisory committee, the members of the SEFCOM Lab, and the folks at Genode Labs for their continued help and support throughout this experience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Today, ARM processors are used to power almost any technology that requires digital intelligence. As of September 2016, ARM has shipped out more then 86 billion chips that are used to securely and efficiently power anything from "smart cities, smart phones, supercomputers, cars, spacecraft, networking base stations and servers" to Internet Of Things devices. [3] Having become a major driving force behind such technologies, ARM has committed to providing the highest levels of security for their processors. To achieve this, ARM introduced the TrustZone hardware security extensions which provide an isolated Trusted Execution Environment (TEE) for modern and future security needs. The resources of this TEE, also known as the Secure World, are separated from the rich, non-secure Normal World at the hardware level. Secure World has privileged access to all system resources, while Normal World, where the rich OS and untrusted applications run, can only access those resources allocated to it.

Since the code in Secure World has the privilege to access Normal World memory and CPU registers, but not vice versa, system integrity checking and malware detection tools can be installed in the Secure World to monitor the potentially compromised Normal World. Therefore, there have been various research efforts into using the TEE provided by hardware to perform the detection of rootkits in the memory [4] [5]. However, all these solutions depend on the ability of the high-privileged TEE to have access over the entire physical memory and the malicious code being present in memory. For example, Trustdump [6] is a forensic toolkit that has a small piece of memory acquisition and integrity checking code stored in Secure World. From the

TEE it can attempt to verify physical memory regions of the Normal World, beyond the reach of the potentially corrupted system.

However, in order to avoid memory introspection tools running in the Secure World (SW), more advanced rootkits are exploiting the fact that Secure World does not have access to the Normal World (NW) cache. [1] In the ARM TrustZone architecture all the resources are tagged with an additional bit (the NS bit) that indicates which world they belong to. Although the Secure World has access to many of the Normal World resources, the presence of the additional NS bit means that the Secure World cannot access the Normal World cache. The authors of CacheKit refer to this as a cache incoherence which they exploit to hide rootkits from introspection tools running in the Secure World. By hiding the rootkit in Normal World cache, it is able to avoid introspection from Secure World tools and therefore evade the primary method of detection used against rootkits. This provides an unprecedented level of stealth which could mean that a compromised system is never able to detect the malicious software running silently in the cache.

First, to ensure that the rootkit is loaded only into cache and not memory where it is susceptible to inspection tools, a technique called Cache-as-RAM is used. [7, 8] However, with this newfound stealth there are some trade-offs that must be addressed. The main goals of a rootkit are to remain:

1. Persistent in the system so as to allow the attacker continued unauthorized access.

2. Undetected by any tools or countermeasures in place.

Cache is volatile memory and is often replaced to cache new memory regions as needed by the system. While this makes CacheKit highly undetectable, it also makes it unstable. If the rootkit cannot remain in the system long enough for the attacker to reach their desired goals then it is useless. Therefore, CacheKit takes advantage of cache locking mechanisms, provided by the ARM Cortex processors at the hardware level, to ensure that the rootkit remains in the cache without being evicted by any replacement policy.

Finally, to ensure that the information of the rootkit can never be recovered, even if the cache is successfully evicted, the virtual memory addresses associated with the rootkit are remapped to unused reserved I/O space. This ensures that a flush will result in the total loss of the rootkit and not it being written back to RAM as normally expected by a cache eviction. Therefore, this novel attack known as CacheKit requires 3 facilities to be able to avoid memory inspection from tools in either world:

1. The ability to employ the Cache-as-RAM method so as not to load the rootkit into RAM.

2. The ability to lock the rootkit in the cache such that it cannot be evicted.

3. The ability to re-map to a new physical address in reserved unused I/O such that any eviction would result in the total loss of all potentially analyzable data. [1]

This new-found level of stealthy attacks leaves many real-time and embedded systems that offer cache locking mechanisms exposed and vulnerable. Therefore, a new defense approach is necessary to continue to provide these hardware facilities for timing and performance sensitive processes. Throughout this thesis, we study possible solutions to determine the best approach to defending against such attacks. After thorough research and experimentation, we design and implement CacheLight, a lightweight approach for preventing malicious abuse of cache locking mechanisms. This novel solution leverages both the TrustZone and Virtualization extensions in the ARM architecture to allow legitimate users to continue to utilize cache locking while giving the Trusted Execution Environment (TEE) the power to ensure system security by controlling and verifying use of said mechanisms. In the next sections of the paper, we provide some background on the TrustZone extensions, the ARM cache and physical memory, and the CacheKit attack as well as related work. With a strong background established, we delve into the process of designing and implementing CacheLight. We implement our defense technique on top of the Genode Operating System. We explore possible solutions using our new environment, but after determining the necessity for valid cache locking, we arrive at the final solution of CacheLight. We present the implementation of CacheLight, and evaluate it in terms of performance and security.

Chapter 2

BACKGROUND: ARM, CACHE AND TRUSTZONE

2.1 ARM Architecture

With the TrustZone Security Extensions enabled, an ARM processor has 9 modes of operations. The modes, respective privileges of software executing in that mode, and security states in which that mode can operate are summarized in Table 2.1 below.

Table 2.1: ARM Processor Modes

Processor Mode	Privilege Level	Security State
User (usr)	PL0	Both
FIQ (fiq)	PL1	Both
IRQ (irq)	PL1	Both
Supervisor (svc)	PL1	Both
Monitor (mon)	PL1	Secure Only
Abort (abt)	PL1	Both
Hypervisor (hyp)	PL2	Non-Secure Only
Undefined (und)	PL1	Both
System (sys)	PL1	Both

The `usr` mode has privilege level 0 and it is where user space programs run. The `svc` mode has privilege level 1 and is where most parts of the kernel run. It is also importation to note that the `mon` mode has privilege level 1 but always runs in secure state only.

At the system level view, the AMRv7 architecture has 16 core registers when Security Extensions are implemented. General-purpose 32-bit registers `R0 - R7`, are the same across all modes. General-purpose 32-bit registers `R8 - R12`, are the same across all modes except FIQ mode. In addition there are three 32-bit registers with special uses. The Stack Pointer (`SP`) and Link Register (`LR`) are banked across all privileged modes except system mode. And finally, the Program Counter `PC` which is not considered a general-purpose register in ARM. Additionally, the `Current Program Status Register` `CPSR` holds processor status and control information, including the current processor mode. The value of this register is copied and saved upon entry in the respective `Saved PSR` by all modes except User and System. This system level view is shown below in Figure 2.1.

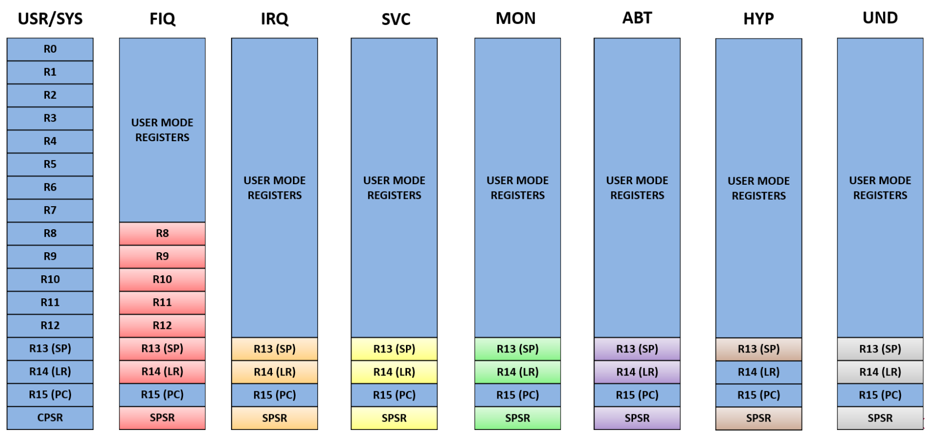


Figure 2.1: ARMv7 Core Registers

The ARM architecture supports coprocessors to extend the functionality of the ARM processor. Each coprocessor has its own set of registers. Coprocessor instructions provide access to sixteen coprocessors in the ARMv7 architecture described as `CP0 - CP15`. `CP15` is called the System Control coprocessor and is reserved is reserved for the control and configuration of the ARM processor system, including architecture and feature identification. [9]

2.2 ARM TrustZone

ARM Processors power billions of networked embedded devices around the world. With a growing Internet of Things and emphasis on security ARM deployed a new technology called TrustZone to provide hardware assisted security. TrustZone is a set of hardware security extensions on the processor, memory, and peripherals that ensure complete system isolation for running secure code. The isolated environment is referred to as the Secure World while the Normal World is the name given to the other environment where the OS and other programs run. Just as how we have user and privileged mode in an operating system, these two hardware worlds interact in much the same way. The Secure World is thought to oversee the Normal World as it has higher access privileges. This means that the Secure World can access most of the resources that belong to the Normal World. Normal World does not have access to any of the resources available to the Secure World. Figure 2.2 shows the isolated environments provided by TrustZone.

However, in hardware these worlds are meant to be completely independent and thus the Secure World cannot access the Normal Worlds cache. To divide the two worlds, `CP15` has a `Security Configuration Register` (`SCR`) with a non-secure (NS) bit that determines the security context of the processor. When the NS bit is set, this indicates that the processor is in the Normal World and when it is cleared, the processor is in Secure World. The two worlds are separated by adding this NS control bit to all system resources. To isolate memory and I/O devices, the NS bit is also added to each read and write channel of the main system bus. For memory, the NS bit acts as a 33rd address bit such that there is a 32-bit physical address space for each of normal and secure transactions. For I/O transactions, the NS bit acts as a flag to indicate if the access attempt has the required privilege. [10] Monitor

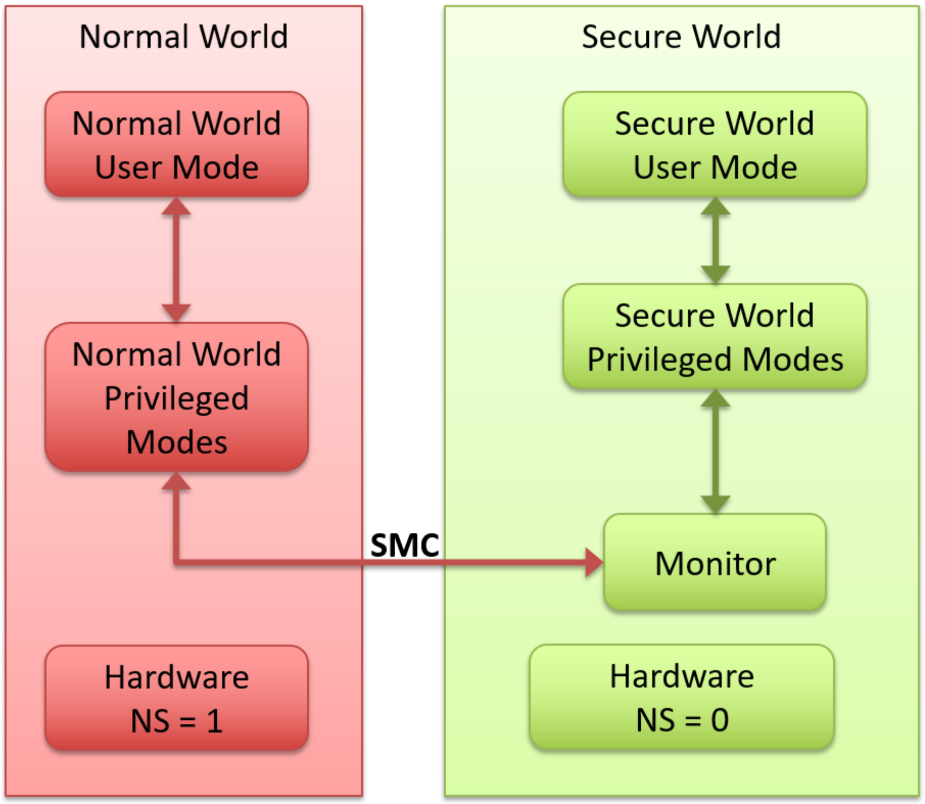


Figure 2.2: ARM TrustZone Security Extensions

Mode is responsible for handling world-switches. To enter monitor mode, the process must execute a Secure Monitor Call (SMC) instruction, at which point monitor mode handles the world switching and determines the necessary function or handler to execute. [2]

2.3 ARM Cache

In general, caches can be divided into four types, based on whether the index and tag bits correspond to physical or virtual addresses.

1. *Physically indexed, physically tagged (PIPT)* caches use the physical address for both the index and the tag. While this is simple and avoids problems with aliasing, it is also slow, as the physical address must be looked up in the translation tables before that address can be looked up in the cache.

2. *Virtually indexed, virtually tagged (VIVT)* caches use the virtual address for both the index and the tag. This caching scheme can result in much faster lookups, since it does not require accesses to the MMU. However, VIVT suffers from aliasing problems, where several different virtual addresses may refer to the same physical address, which severely complicates the sharing of caches.

3. *Virtually indexed, physically tagged (VIPT)* caches use the virtual address for the index and the physical address in the tag. They are faster than PIPT caches as the cache line can be looked up in parallel with the TLB translation, however the tag cannot be compared until the physical address is available. While they still face aliasing problems, the physical tag bits make sharing more manageable.

4. *Physically indexed, virtually tagged (PIVT)* caches are generally not implemented.

In general, the ARM L1 instruction cache is VIPT for greater speed and since instructions generally are not shared. However, the L1 data cache and any lower level caches are generally PIPT, with some options for implementing them as VIPT. [11]

With the implementation of TrustZone technology, the processor cache is also extended with an additional tag bit. This tag bit is used to record which world made the memory access, or which world the line of cache belongs to. This makes it so that the processor always knows which lines of cache belong to which world and cache flushing between world switches is not necessary. The two worlds can evict each others lines in the cache as needed. [9] However, it is possible to prevent cache lines from being evicted by locking them. While this allows for performance optimization, it also gives rise to the cache incoherence which CacheKit exploits to hide from memory inspection tools. After the virtual address and physical address translation has occurred in the Translation Lookaside Buffer, level 1 cache stores the memory data of the physical address and keeps the NS bit to indicate which world it belongs to. This NS bit is set by hardware and it is not directly accessible by system software. [12]

Through several experiments on the i.MX53 development board, Zhang et. al determine that the addition of TrustZone introduces a cache inconsistency where each world cannot access the other worlds cache content. Therefore, while the Secure World cannot access the Normal Worlds cache content, though it can access the Normal Worlds RAM. This is because the architecture tries to optimize the performance of the implementation by separating cache lines based on the processor execution mode with the NS bit. However, this means that cache lines in the Normal World are only visible to the Normal World. While this ensures that the two worlds are independent, this also means that this is a resource in the Normal World that security and forensic tools in the Secure World cannot inspect unless it is flushed into RAM.

Chapter 3

CACHEKIT ATTACK

CacheKit is a new type of rootkit that exploits a cache incoherence in the ARM TrustZone architecture. [1] Dividing the two worlds completely results in a cache incoherence where the contents of Secure and Normal World are different even if they map to the same physical address. CacheKit exploits the fact that even though the Secure World can access the memory of Normal World, the two worlds are separated such that they cannot gain access to the others cache. There are three major steps in establishing CacheKit: Loading, Locking, and Hiding. First, a technique known as Cache-as-RAM is used to ensure that the rootkit is loaded only into cache of the Normal World where it can avoid detection from the Secure World. [7] Then, the ARM hardware support settings are exploited to keep the code persistent in cache as long as possible. Finally, the translation tables are modified such that the malicious code in cache maps to unused I/O addresses in physical memory so that if cache content is flushed to RAM for inspection, the data is simply lost. This ensures that even if the rootkit were to be flushed into memory for any reason, any trace of the malicious code would be lost. [1]

3.1 Cache Incoherence

Through several experiments on the i.MX53 development board, the authors determine that the addition of TrustZone introduces a cache inconsistency where each world cannot access the other worlds cache content. Therefore, while the secure world cannot access the normal worlds cache content, though it can access the normal worlds RAM. This is because the architecture tries to optimize the performance of the implementation by separating cache lines based on the processor execution mode with the NS bit. However, this means that cache lines in the normal world are only visible to the normal world. While this ensures that the two worlds are independent, this also means that this is a resource in the normal world that security and forensic tools in the secure world cannot inspect unless it is flushed into RAM.

3.2 CacheKit: Loading

Once the cache incoherence in TrustZone has been confirmed it can be exploited to conceal a rootkit within the normal world cache that avoids memory inspection techniques. In the ARM architecture, the only way to read or write a cache line is to have the processor read from or write to virtual memory. Therefore, to load the rootkit into cache the CacheKit module must first enable caching on memory. This is done by setting the paging table memory attribute as WriteBack. The WriteBack configuration ensures that load register (LDR) instructions trigger a cache line fill. The key of cache loading is to ensure that data is loaded to cache and cache only, the rootkit should not be stored in RAM at any point as it would make it visible to memory inspection.

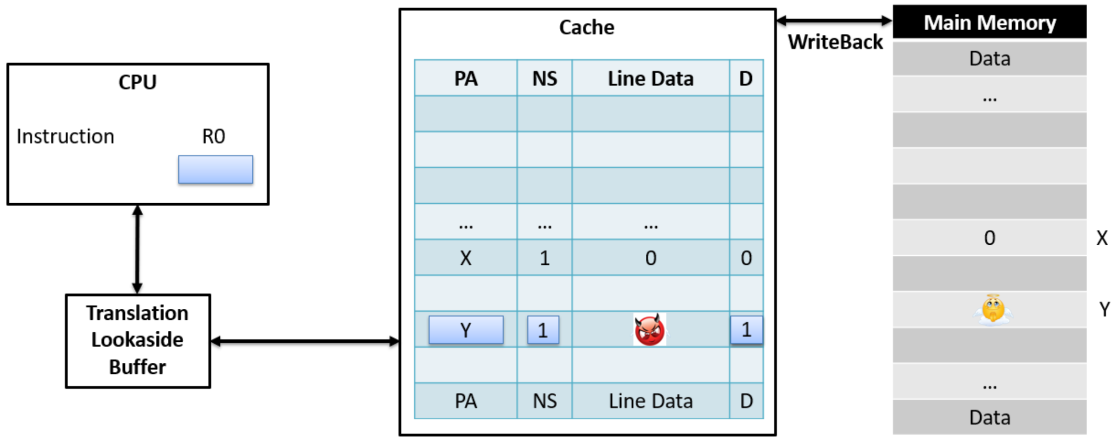


Figure 3.1: Exploiting the Cache-as-RAM Technique

Figure 3.1 shows how, using the Cache-as-Ram technique, it may be possible to have code in cache that is not consistent with what is in main memory, thereby allowing an attacker to modify benign code in the cache with malicious code without it being written back to memory right away.

3.3 CacheKit: Locking

The Cortex-A8 processor allows system software to lock up to seven cache ways out of the total eight ways. This ability to maintain the code persistently in cache is essential as cache is very volatile. Once the rootkit is loaded, it should remain in memory as long as possible so an attacker can maintain control over the infected system. While using Cache-as-RAM enables us to better hide the malware it also means that the code can survive for a reasonable amount of time to make it useful. First, the cache corresponding to all the memory addresses to be locked in cache will need to be flushed out. Second, the cache way to be used is unlocked and the other ways are locked. This means that any cache fills made by the LDR and STR instructions will be made to the way that has been designated for the rootkit. Lastly, once the rootkit has been loaded into cache the way that has been reserved for the rootkit is locked and the rest of the cache is unlocked. Consequently, the implementation of Cache-as-RAM to store the rootkit gives it exceptional stealth as it can no longer be detected by memory inspection tools. However, there is a trade-off in storing the rootkit in cache. Cache is very volatile and even though cache lines can be locked with hardware control, they will still respond to cache maintenance instructions if they are called. Therefore, when a cache flush is called the contents of the rootkit will indeed be written out to memory. While mapping the cache to I/O address space does mean that an attacker can maintain stealth by destroying the data since no backup memory exists, it also means that they lose the rootkit if it is ever flushed from cache. Since the effectiveness of a rootkit depends on how long it can remain hidden in the system this is a major trade-off between stealth and persistence. Therefore, for CacheKit to be most effective it is imperative to have an environment where data can persist in cache. Upon conducting a search of the kernel source code the authors determine that there is only one function that the kernel could call to clean the entire cache, thereby evicting the rootkit. However, it is never used in the uniprocessor deployment which makes the i.MX53 particularly vulnerable to a CacheKit attack as persistence can be guaranteed at least until power cycling.

3.4 CacheKit: Hiding

Having successfully loaded and stored the rootkit in cache, CacheKit must now address the two main issues of remaining concealed. The first problem is that even when locked, the cache lines they are still responsive to cache maintenance instructions. As stated, ARMv7 provides various cache maintenance instructions that can cause the contents of the cache to be written out to memory. This is dangerous because if the malicious code is flushed to memory then it becomes detectable to memory inspection techniques. Similarly, the second problem has to do with writing back to memory and introspection from the normal world kernel. Detection methods that sequentially map each physical page into the kernel memory space would still be able to read the cache. Therefore, the idea of mapping the cache lines to unused physical I/O address space is proposed to resolve these issues with direct cache locking. The authors refer to this newly mapped space as CacheKit Space.

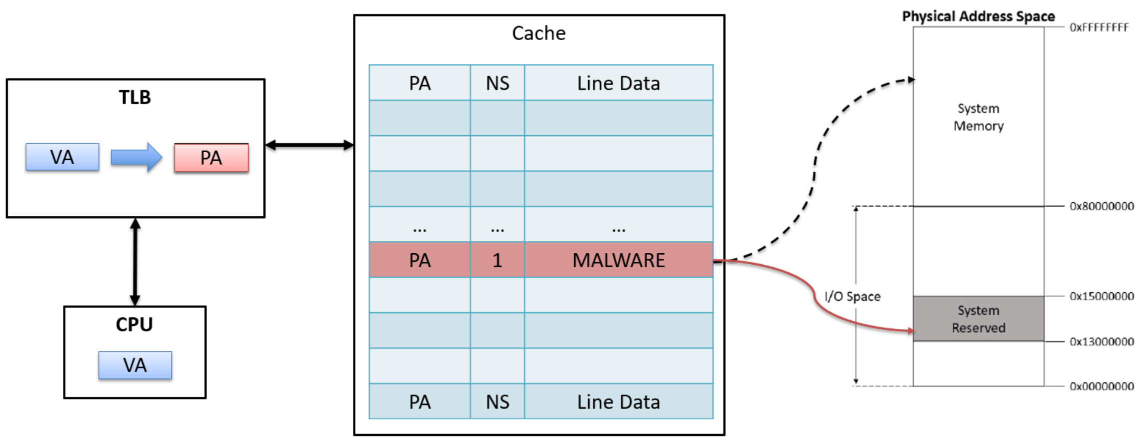


Figure 3.2: Hiding the Rootkit in the Cache

As shown in Figure 3.2, the key is to find the memory allocation of the system in question and determine which part of the address space is used for I/O devices. In 32-bit ARM the convention is to have the first 2GB of memory mapped to I/O space. Furthermore, in the i.MX53 development board it is found that the address range between 0x13000000 to 0x15000000 belongs to reserved, unused I/O space. Therefore, any access requests to this area are redirected to peripheral bus. CacheKit takes advantage of this and carves out this space for potential flushes by reconfiguring it as memory space using the Cache-as-RAM technique. Now, all read and write operations are redirected to the cache of the processor. This newly configured CacheKit Space is neither backed by RAM nor any real I/O device. Therefore, should the malicious code ever be evicted by a cache clean or flush there is no physical device that will respond to the memory write. This ensures that upon cache eviction, instead of being written to memory where it can be discovered the malicious code is simply lost with no error, response, or detection, as though it had never existed. This allows the intentions and operations of the rootkit to remain unknown and untraced, perhaps for future re-loading.

Through several tests the authors are able to show that the methods of loading, locking and hiding the rootkit in cache are effective. Several tests indicate that CacheKit is not detectable by memory inspection tools run in the secure world. Additionally, the DMA-based hardware forensic tool is also unable to detect the rootkit in the cache. Therefore, the methods employed by the CacheKit module effectively exploit cache incoherence behavior in TrustZone to hide the rootkit from memory inspection techniques.

Chapter 4

CACHELIGHT

In this chapter of the thesis, we discuss various solution approaches investigated. Generally, security measures can be broken down in to three categories; detection, prevention, and response. The focus of this thesis is on detection and prevention. We compare the benefits and limitations of each approach and eventually conclude that the approach we name CacheLight provides the best solution.

4.1 Naïve Approaches

4.1.1 Prevention

First, we focus on preventing rootkits from hiding in Normal World cache. According to the ARM Technical reference manual for the Cortex-A8 [2], the CL determines whether or not cache locking can be done in the Normal World. If the CL is set (to 1) then cache locking is available in both worlds. However, if the CL bit is cleared (to 0) then cache locking is only available in the Secure World. Attempting to use registers associated with cache locking, namely the `L2 Lock Down Register`, an undefined instruction exception is thrown. Therefore, this provides a simple and direct solution to preventing CacheKit attacks by disabling the cache locking capabilities in the Normal World and only allowing trusted applications running in privileged SW levels to lock cache entries. While this would prevent such attacks entirely because the rootkits would no longer be able to remain persistent in cache, it also limits a lot of what NW users can accomplish in their applications.

Therefore, disabling cache locking may not be an option. Many modern processors feature cache locking mechanisms to allow for finer control of cache eviction policies and thereby improving the cache hit rate. This can have a considerable impact on the performance of a system if managed correctly. [13] For this reason, a wide variety of processors across different manufacturers and processor families offer this option. Some of ARMs Cortex and ARM11 processors allow for way locking in which locking is available at the granularity of ways of a set-associative cache. Line locking is a more fine-grained approach where it is possible to have a different number of locked lines in different sets of the cache; Intels XScale, the ARM9 family, and BlackFin 5xx family processors support this kind of locking mechanism. [14]

Originally, cache locking was designed to offer timing predictability for real-time systems, where techniques focused mostly on improving worst-case execution time. However, recent research has expanded this use to better performance of general embedded systems by improving average-case execution time. While the improvement can be significant, there is also the risk that an improper implementation will end up negatively impacting performance, therefore careful management is essential to fully realize the potential of cache locking. [15] Several techniques have been developed to try to use cache locking to optimize performance as categorized by Mittal. [15]

Many different mechanisms have been developed, each with their own benefits and limitations, to try to leverage cache locking to the fullest. These include both dynamic and static cache locking as well as partial and full cache locking. [16] There are also works that focus on guaranteeing a minimized predictable average worst-case execution time for embedded systems by exploring different profiles. [13, 14] In fact, the i.MX53 was chosen because it features an ARM Cortex-A8 processor which supports cache locking.

Considering how important cache locking can be in both real-time and embedded systems across so many processors we cannot assume that disabling the mechanism is an option.

4.1.2 Detection

Therefore, we move to detecting rootkits hidden in the cache.Given that it is most likely a part of a real-time or general embedded system that depends on cache locking to give critical processes timing predictability or the ability to meet stringent performance requirements, we cannot move this functionality to the Secure World. Not only would it affect the timing with the need for world switching but it would also greatly broaden the Trusted Code Base, defeating the purpose of having a small, isolated, trusted environment. Therefore, it is necessary to enable cache locking in the Normal World and thus we must be able to detect malicious attempts my a potentially compromised Normal World OS.

Originally, we had considered that in order to detect an attack, we would need not only the attempt to lock the cache but also the remapping of a virtual address to a physical address that does not correspond to system memory. That is, these two fundamental necessities of CacheKit would be enough to determine that a malicious user is attempting to hide code in the Normal World cache. CacheKit prevents its data from being retrieved for forensic purposes by mapping to the physical region of reserved unused I/O so that if the cache is ever flushed, the malicious code is simply lost and cannot be analyzed. However, these two functionalities together make for a very specific signature that can most definitely signal malicious activity. This is because while some time critical programs may need instructions or data to be locked in th cache, we would never need anything outside the System Memory Region to be locked.

The first step is to detect if Normal World has attempted to lock the cache by reading the value of the `L2 Lock Down Register`. Anything other than 0 would indicate that one or more cache ways are locked. If we are able to then find a page table entry that maps a virtual address to a physical address outside of the System Memory Region, we would be able to detect a rootkit hiding in the cache. This would require a page-table walk, because we have to check all entries to ensure completeness. While a page-table walk is expensive, we would only need to execute it when a lock is detected. In this manner, we would then be able to force a cache flush or invalidation to trigger a write back to memory. To ensure that we are able to retrieve the malicious code we also have to modify the virtual address that has an invalid mapping to a new, valid memory region that we have reserved from Secure World for memory inspection. However, we find that this approach will be ineffective in retrieving the malicious data in the cache because even if we are able to re-map the virtual address to a valid physical address, the L2 cache in the ARM Architecture is Physically Indexed, Physically Tagged. This means that once the re-mapped address has been loaded into the cache, it is already too late retrieve the data. This approach would only work with a processor that supports both cache locking and VIVT caches.

In the ARM Architecture for the Cortex-A8, the L2 cache is Physically Indexed, Physically Tagged (PIPT). [2] This means that the tag and index bits stored in the cache to identify which memory address the cache line maps to are taken from the physical address and not the virtual address.

As shown in Figure 4.1, even if we are successful in re-mapping the virtual address to a valid physical address, once we flush the cache the write to memory will be done according to the tag and index bits stored with the respective cache lines. Therefore, once the rootkit has been stored in the cache with the modified physical address that points to reserved, unused I/O, any eviction will always be done to this location.

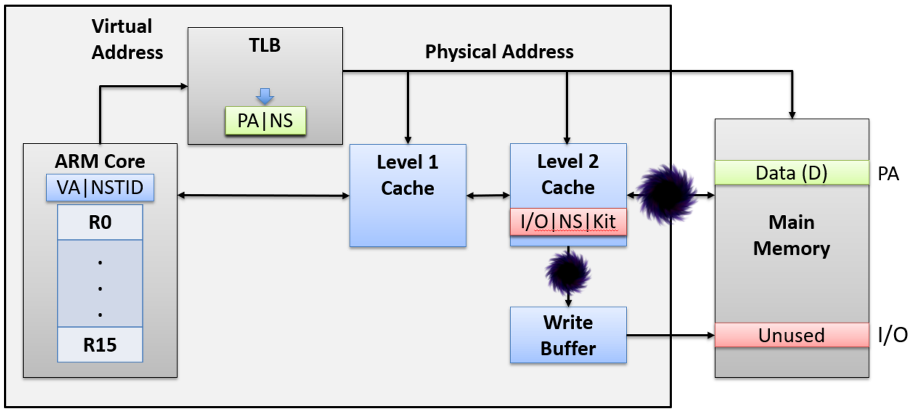


Figure 4.1: PIPT Caches Prevent the Retrieval of Hidden Data

CacheKit creates a "black-hole" situation where once it is in the cache, there is no retrieving it because there is no way to access the tag and index bits of the cache other than with perhaps a JTAG Debugger. Therefore, the only solution left is to develop a defense mechanism to prevent rootkits, or any malicious code, from being loaded into the cache in the first place. This makes a detection solution considerably difficult. However, by leveraging TrustZone technology, prevention is still possible.

4.2 CacheLight Approach

In this section of the paper we present our CacheLight approach, a defense mechanism for preventing malicious software from inhabiting the cache for any significant period of time. We leverage the TrustZone Security Extensions to ensure that even if an attacker is able to briefly compromise the Normal World kernel, they will not be able to leave behind any malicious code in the Normal World Cache. The main idea is to go expand on the most simple solution of disabling the ability to lock the cache by clearing the CL bit. However, we still want user programs that have time-sensitive operations in real-time and embedded systems to be able to make use of the cache locking mechanism. Therefore, we implement CacheLight as a Secure World kernel function that can be called from Normal World kernel to have the Secure

OS perform the locking for the Non-Secure OS. This means that while users can still take advantage of the benefits of cache locking for legitimate purposes, the Secure OS can have better control over what gets locked in the cache and perform integrity checking to prevent malicious code from being loaded in. In order to achieve this, we have to switch from a data abort to an SMC which the Normal World can call and the Secure World can then handle.

4.2.1 Passing Arguments to Secure World

The first step is to determine what arguments the pass with the SMC to Secure World. In order to service the call, the SMC Handler in Secure World must know:

1. OP Code: The operation code to determine what the SMC is for and properly handle the request.

2. VA: The virtual address that corresponds to the data that the NW wants to lock in cache.

3. LockDownReg: the `L2 Lockdown Register` value to be used. This determines which cache ways the Normal World wishes to lock.

4. Size: the size or amount of data to be loaded into the cache from the base address (VA).

With these three arguments we have all the information we need to load and lock the data into the cache on behalf of the Normal World. More importantly, the data must first be loaded into physical memory by the Normal World, which ensures that whatever is being loaded into the cache is consistent with what appears in RAM and therefore exposed to memory inspection tools.

4.2.2 CacheLight *Work-flow*

Once we have establish the necessary parameters to pass to Secure World, we can detail the necessary work-flow, or steps, of the approach as shown in Figure 4.2:

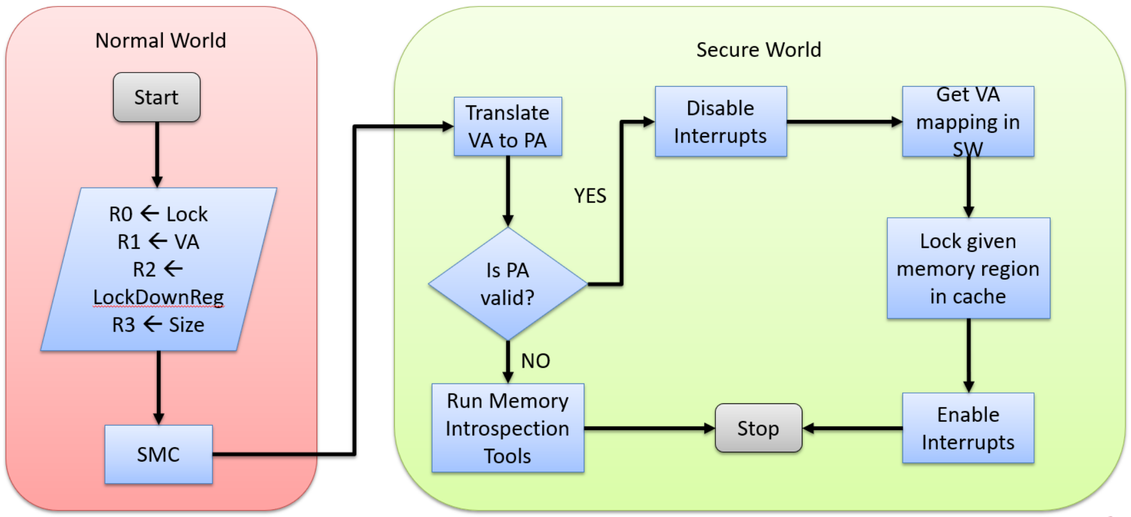


Figure 4.2: Work-flow Diagram for CacheLight

Secure World has all the necessary information to first verify that the request to lock memory is valid by ensuring only memory regions allocated to Normal World are being requested. Furthermore, all the data must first be loaded into main memory by the Normal World, and is therefore available to inspection, should Secure World find an invalid PA is given. Otherwise, it can then go through the process of servicing legitimate requests and locking the requested memory in cache. The steps for this case are detailed in the following sections.

4.2.3 *Virtual to Physical Address Translation*

Virtual address translation is provided by way of the Memory Management Unit. In ARMv7 the MMU works with the L1 and L2 memory system to translate virtual addresses to physical addresses and controls accesses to and from external memory. Therefore, it is enhanced with security extensions and multiprocessor extensions to provide address translation and access permission checks. Virtual-to-physical address mappings and memory attributes are defined in main memory as page tables; each World has it's own set of page tables and TLB identifiers to remove the requirement for context switch TLB flushes. [2]

Therefore, the first step in preventing rootkits from being loaded into cache is to translate the given VA to the PA. The ARMv7 Architecture provides VA to PA address translation operations using registers from `CP15`. There is a `Physical Address Register (PAR)` that holds the PA after a successful translation or the source of the abort after an unsuccessful translation. It is a read/write register banked in Secure and Non-secure states and accessible in privileged modes only. [2] Figure 4.3 shows the format of the `PAR` register.

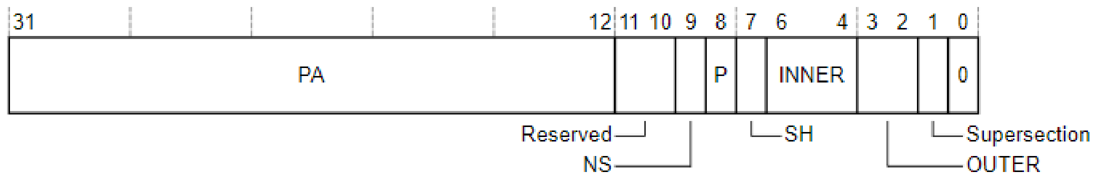


Figure 4.3: Physical Address Register in ARM [2]

Notice that bits [31:12] contain the the physical address after a successful translation. The rest of the bits are taken directly from bits [11:0] of the virtual address. This gives us a complete PA to use. To make use of this register for our address translation we do VA to PA translation in the other Secure or Non-secure state. The purpose of the VA to PA translation in the other Secure or Non-secure state is to translate the address with the current virtual mapping in the Non-secure state while the core is in the Secure state. [2] To access the VA to PA translation in the other Secure or Non-secure state, we write CP15 c7 with Opcode2 set to 4 for privileged read permission. In this manner, the given virtual address will be translated using the Normal World's translation tables and we can retrieve a PA to work with.

4.2.4 Verifying Memory Contents

Once we have the physical address we are able to determine it is a valid address within the system memory. We can check if the PA maps to I/O memory region, in which case we know we should not allow such an address to be locked in the cache. Depending on the implementation, it is also possible to check that the PA is withing the memory region allocated to the Normal World by the Secure World for a much better, tighter security check. At this point we can flush the cache and run the memory inspection tools of the Secure World to retrieve and analyze the malicious code that the attacker attempted to leave behind. Since in this scheme only the Secure World is allow to perform cache locking, we are able to ensure that any memory that will remain in the cache does not differ to what is in RAM and thus we can be certain that there is no incoherence from what we are inspecting in the RAM and what the system is executing in the cache.

On the other hand, if the request is for a valid physical address then the secure world must be able to perform the loading and locking of the requested memory region into the specified cache way. The details of this case are discussed in the following sections.

4.2.5 Enabling and Disabling Interrupts

The first step is to disable interrupts so that we cannot be pre-empted. To lock the given memory, and only the given memory, in the given cache way(s) the locking process must not be interrupted. Therefore it is imperative that this code be within a non-preemtible critical section. Once the locking is complete, Secure World can enable interrupts again.

4.2.6 Mapping Normal World Memory to Secure World

With the virtualization extensions active in both worlds, it is necessary to map a virtual address in the Secure World to the given physical address from the Normal World. This means that the Normal World memory must be mapped in the Secure World as well. Memory that both worlds have access to is referred to as World-Shared Memory. World-Shared Memory is denigrated as non-secure since NW can only make non-secure accesses while SW can make both secure and non-secure accesses. Now that valid VA entries exist for Normal World physical addresses, Secure World can use these to load the requested memory region and thereby create the cache line entires on behalf of the Normal World. Given the PA that the Normal World would like to lock, a helper function is defined to build the offset within the physical Normal World memory and add that offset to the virtual start address of the mapping within the Secure World kernel, returning the Secure World VA for the specified PA. Since the processor is in Monitor Mode, this VA can be used to create cache entries tagged as Non-Secure that will generate cache hits when the Normal World attempts to access said memory, effectively allowing the Secure World to create locked cache entries for the Normal World.

4.2.7 World-Shared Memory

As mentioned in Chapter 1, all addresses are tagged with an additional NS bit to indicate what mode the entry belongs to. Normal World is only able to access entries tagged with an NS bit equal to 1. Therefore, when we load the data into the cache we have to ensure that it is tagged with NS = 1 so that Normal World will have a cache hit when attempting to access the physical address that corresponds to said data.

Figure 4.4 shows how the memory system of a theoretical ARM processor might handle the state associated with Security Extensions when accessing the memory system. [12]

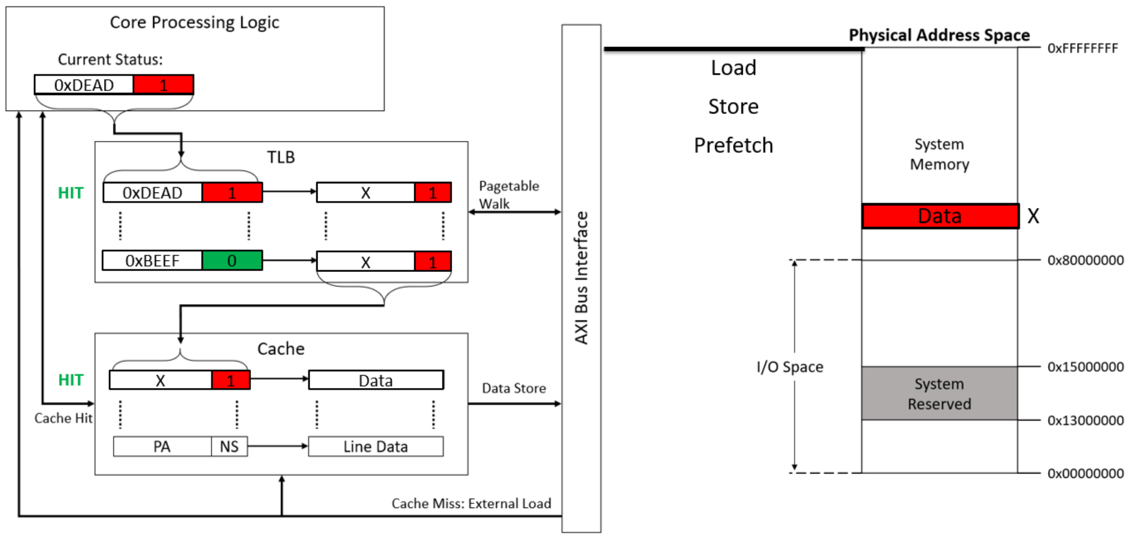


Figure 4.4: Memory System for a Theoretical ARM Core

First, the core processing logic attempts a data load, a data store, or an instruction prefetch. The hardware passes the Virtual Address (VA) and the current world (Non-Secure Table Identifier, or NSTID) to the TLB to enable it to perform address translation. the NSTID is passed by the hardware and depends on the current state of the processor, not the NS bit. In this case the NSTID will represent a Secure World entry. [12]

Then, the TLB loads the physical address and the NS-bit associated with the VA and NSTID it was passed, performing a page-table walk and forcing NS=1 if NSTID=1 if necessary. The translation tables are responsible for keeping track of the NS bit and whether the VA resolves to a secure or non-secure physical address. [9] The TLB then passes this information to the cache to perform the actual data or instruction access. If the Secure World has mapped the non-secure memory containing the data the Normal World wants to lock in its translation tables, then the Secure

World can directly access the non-secure cache lines. Therefore, a Normal World application can pass data to the Secure World though any level in the cache hierarchy. This enables a high performance system in comparison to solutions that require cached data to be flushed out of the cache and in to external memory. [12]

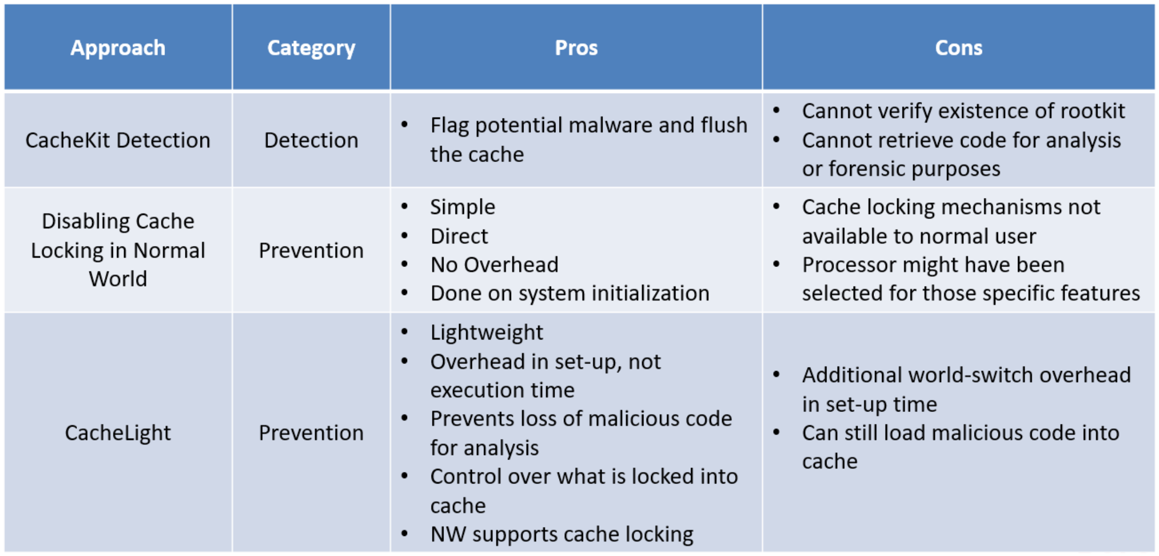
Therefore, when we later on perform the loading and locking of memory the entires of the cache will we tagged with an NS bit of 1 rather than zero. This means that when the Normal World attempts to access this address that it requested Secure World to load into the cache for it, it will generate a cache hit. Normal World will find a cache entry that matches the PA translation with an NS bit of 1, indicating that this memory is designated for Normal World use and will return the data from that cache line. This also means that Secure World can gain access to the Normal World cache if it creates the necessary translation table entries by implementing World-Shared Memory

4.2.8 Locking NW Memory Into Cache From SW

Finally, we can now perform the loading and locking of the memory in the Secure World exactly as it would have been done in the Normal World. We pass the virtual address, the value to use for the `L2 Lockdown Register`, and the size of the data to load as arguments. Therefore, Secure World has all the information required to load and lock data to the requested cache ways. By implementing World-Shared Memory, the Secure World can issue non-secure memory accesses that create non-secure cache entires for the Normal World. When control is returned to the Normal World, the legitimate process will be able to access the time-critical section locked in the cache as intended. Therefore, this mechanism gives the Secure World a lot of power and control over what gets loaded into the cache, allowing it to ensure that there is no incoherency with what is stored in memory and what is stored in Normal World cache.

4.3 Comparing Approaches

To the best of our knowledge, there have been no other defense mechanisms proposed against these kinds of attacks. Therefore, to evaluate the best solution, we provide a comparison of the three approaches introduced here.



Approach	Category	Pros	Cons
CacheKit Detection	Detection	• Flag potential malware and flush the cache	• Cannot verify existence of rootkit • Cannot retrieve code for analysis or forensic purposes
Disabling Cache Locking in Normal World	Prevention	• Simple • Direct • No Overhead • Done on system initialization	• Cache locking mechanisms not available to normal user • Processor might have been selected for those specific features
CacheLight	Prevention	• Lightweight • Overhead in set-up, not execution time • Prevents loss of malicious code for analysis • Control over what is locked into cache • NW supports cache locking	• Additional world-switch overhead in set-up time • Can still load malicious code into cache

Figure 4.5: Comparison of Different Defense Approaches

We conclude that, while there is room for further work, the most robust and effective solution is the CacheLight approach. In the following chapters we implement and evaluate CacheLight in terms of security and performance.

Chapter 5

CACHELIGHT IMPLEMENTATION

5.1 Genode: A Secure World OS

To defend against this new type of attack we first replicate the original prototype scenario on the i.MX53 Development Board. Once functionality was verified, we moved to deploy the defense environment. The first step is to have a Trusted Execution Environment (TEE) in the Secure World. After careful consideration we decided to use the open-source Genode Project which supports the i.MX53. Genode provides support for various Normal World operating systems, hardware platforms, and scenarios. The scenario we choose was designed for the i.MX53 with TrustZone hardware capabilities. As shown in Figure 5.1, Genode runs in the Secure World and deploys a VMM in which it then boots a modified Linux 2.6.35 simple kernel in the Normal World.

The Linux kernel was modified such that certain kernel functions could handle world switches using an SMC call to the Secure World. The CacheKit attack module can then be cross-compiled for the new target kernel in the Normal World and deployed on the new Genode environment. [17]

5.2 Building and Deploying The Environment

Using Genode 18.02, we build the Genode Truztone Virtual Machine Monitor (TZ-VMM) scenario for the i.MX53 board. This build generates an bootable image for the target board as specified in the configuration file. The next step is to obtain the necessary bootloader provided by Genode Labs and generate the boot image.

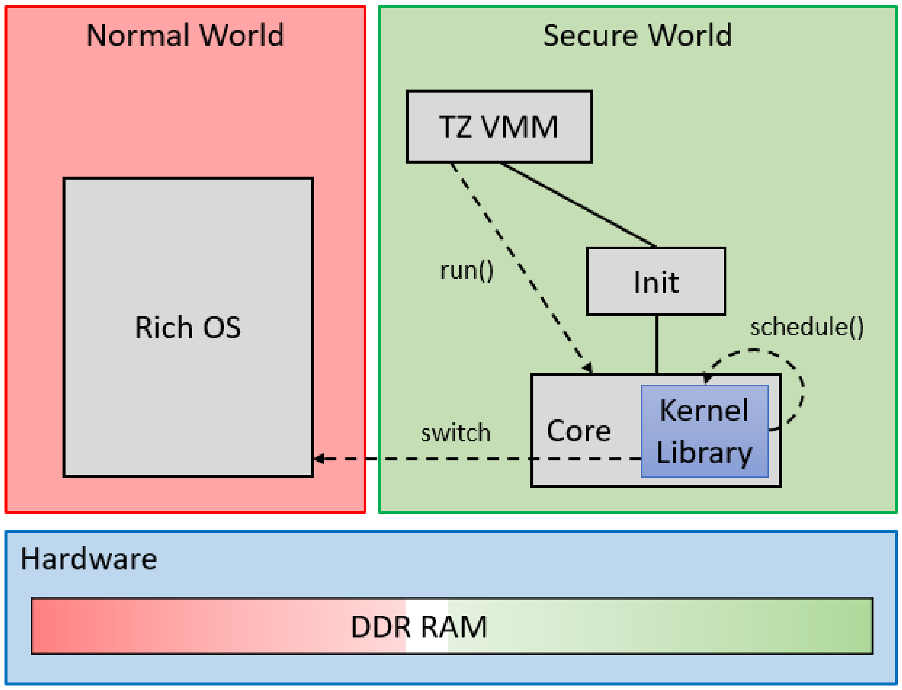


Figure 5.1: TrustZone VMM Scenario in Genode

Finally, a bootable SD card is prepared for the board using both the boot image and the Genode image generated earlier. Now we have the Genode TZ-VMM running on the i.MX53 development board. The details of using and understanding Genode and all of the different scenarios can be found in their Foundations Book. [18]

5.3 Deploying the CacheKit Attack

We replicated CacheKit on the i.MX53 development board, which features a single ARM Cortex A8 processor. The project is implemented as a kernel module that once installed on the board can use the cache manipulation tools in ARM to successfully load, lock, and conceal the rootkit in cache.Next, we must be able to cross-compile the CacheKit attack module against the new modified target kernel running in the Normal World. Unfortunately, the modified kernel is provided as a pre-compiled image in the Genode repository. Therefore, we obtain the source code of the modified kernel and create our own build. This then gave us a target to cross-compile the CacheKit attack module against to be able to deploy it on the new environment.

5.4 Disabling Locking

Now that we have our prototype set up we deploy the attack module and find that it is unable to perform the hiding functions in the new environment. After examining the Genode source code for the Secure World we find that upon initialization the CL bit of the `Non-Secure Access Control Register (NACR)` is cleared as shown in Figure 5.2.

```
/* Non-Secure Access Control Register */
ARM_CP15_REGISTER_32BIT(Nsacr, c1, c1, 0, 2,
/* Co-processor 10 access */
struct Cpnsae10 : Bitfield<10, 1> { };
/* Co-processor 11 access */
struct Cpnsae11 : Bitfield<11, 1> { };
struct Ns_smp   : Bitfield<18, 1> { };
);
```

Figure 5.2: Genode Does Not Enable Locking in Normal World by Default.

Therefore, when the kernel module attempts to lock the cache, an undefined instruction exception is thrown by the system as shown in Figure 5.3.

As shown in Figure 5.3, when the attack module attempts to access the `L2 Lock Down Register` to lock down a cache way, an undefined instruction exception is thrown and the module terminates. Notice also that when we read the NACR the value is given as initialized Genode where only access to co-processors 10 and 11 are

```
CacheKit: Cache Module Loaded.
Printing System Config:
CL (bit 16) value: 0x00000C00
Attempt to set CL bit:
CL (bit 16) value: 0x00000C00
Read L2 Lockdown Register: 0x00000000
Write L2 Lockdown Register:
Internal error: Oops - undefined instruction: 0
Modules linked in: CacheKit(+)
CPU: 0             Not tainted
PC is at writeL2LockedDownRegister+0x0/0xc [CacheKit]
```
Figure 5.3: Cache Locking Disabled in Normal World

enabled. [17] This means that the CL bit is set to 0 and thus the attack module is rightfully denied the ability to lock down the cache. This is a very simple and direct method for preventing rootkits from hiding in the Normal World cache to avoid memory introspection.

5.5 Detection Attempt

For the detection implementation, we enable cache locking in the Normal World for Genode. We then deploy the CacheKit attack module and attempt to detect when the Normal World locks the cache and determine whether or not the attempt is malicious. The first step is to be able to determine if the Normal World has locked the cache.

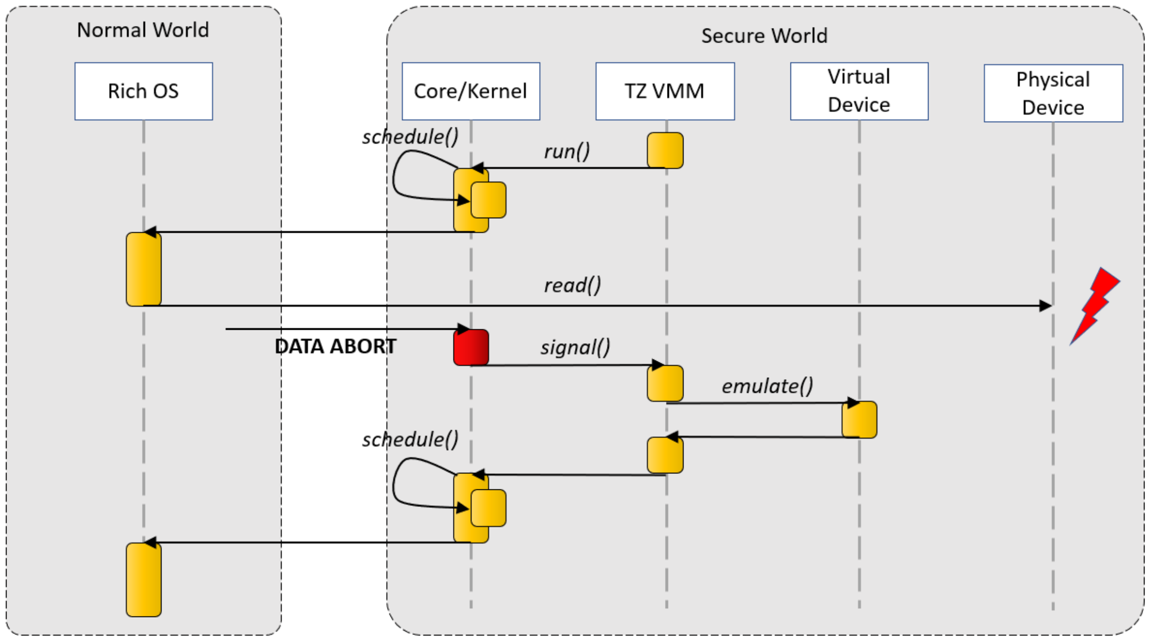


Figure 5.4: World Switching on Data Abort Exception

In the Genode architecture, a world switch from Normal to Secure World happens only when an SMC call is made or when a Data Abort Exception occurs. A data abort exception occurs when the Normal World attempts to access a physical memory region associated with a device that the rich OS in the Normal World was not given access to. [17] The data abort triggers a switch back to Secure World where the trusted OS can produce a signal to handle the exception and emulate the device if access is warranted for the Normal World. The signal handler executes at a privilege level in the Secure World. The execution in the trusted kernel following a data abort, highlighted in red in Figure 5.4, provides a great opportunity to check Normal World behavior with ease of testing. Therefore, we chose to deploy our detection code upon a data abort since the `L2 Lockdown Register` requires privileged access. Before the signal is sent, we check for locked cache lines by reading the `L2 Lockdown Register` and are able to determine is the cache is locked. The signal is sent only if the value of the register is zero. [2] Any other value would indicate locking and we must handle and verify this before we allow the Normal World to continue. However, we are unable to verify malicious code was indeed hiding in cache or retrieve any of the data for analysis because the cache is PIPT.

5.6 Deploying the CacheLight Defense

The Normal World kernel was modified such that certain kernel functions could handle world switches using an Secure Monitor Call (SMC) to the Secure World. According to their documentation, there were six different SMC's added to the Linux kernel. [19] However, upon closer inspection of the current release, Genode 18.02, there are actually only 4 SMC's that are implemented. Furthermore, they do not follow the ARM SMC Calling Convention [20], but rather implement their own, simpler version of handling SMC's. In Genode, an SMC is still used to generate a synchronous exception that is handled by Secure Monitor code running in Exception Level 3 (EL3). The Secure Monitor Mode handles the world switching and then hands off the handling of the exception to the Virtual Machine Monitor running as an unprivileged user-level component in Genode. However, instead of using the ARM Calling Convention, since there are only four different calls implemented, the function to perform is simply passed as an argument in register `R0`. Registers `R1-R10` are then used to pass any necessary arguments to the Secure World. Upon a world-switch the Monitor will save the state of the Virtual Machine, including these registers, so that the Secure World can have access to them. The VMM then simply checks the value in R0 to determine what function the Normal World needs it to perform for the SMC and expects any relevant arguments in the other registers. [19] This makes it fairly simple for us to add and define our own, fifth, SMC for a request to Secure World to lock memory in the cache. The table below shows the different SMC's that Genode had already implemented with the addition of our own "LOCK" SMC with code 4.

Table 5.1: SMC Implementation in Modified Linux

SMC Number	Function
0	FRAMEBUFFER
1	INPUT
2	SERIAL
3	BLOCK
4	LOCK

As shown in Table 5.1 our SMC has a function code of 4, this is passed through register `R0` and is used to determine what function the Secure World needs to perform and the respective arguments in the remaining registers. In our case, we pass the following arguments:

1. R1 is the virtual address to be loaded and locked in the cache.
2. R2 contains the lock down register value to be used. This determines which cache ways the Normal World wishes to lock
3. R3 the size or amount of data to be loaded into the cache from the base address in R1

With these three arguments we have all the information we need to load and lock the data into the cache on behalf of the Normal World. More importantly, we have the location of the data before it gets stored in the cache which allows us to regulate what can be locked into the Normal World cache. Once the request is handled by Genode, it restores the VM's state and switches back to Normal World operations.

After we have implemented an SMC for CacheLight in the Genode core, we need to define what happens once it is called by the Linux kernel. The SMC would be called whenever the Normal World has some code that needs to be locked in the cache. The Normal World would load the operation code, virtual address, value for the lock down register, and size of data to be loaded into the respective registers and perform an SMC. Once Genode reaches the exception handling for the call, we need to be able to create an entry in the cache with the data requested by the Normal World that is accessible by Normal World. The general algorithm for CacheLight is detailed below:

Algorithm 1 CacheLight Algorithm

```
1: function HANDLE LOCK SMC( VA, LockReg, size)
2:     $PA \leftarrow virt\_to\_phys\_in\_other\_world(VA)$
3:     if $PA$ < NONSECURE_RAM_BASE || $PA$ > NONSECURE_RAM_END then
4:         Run Memory Introspection Tools
5:     else
6:         Disable Interrupts
7:         Get VA mapping in Secure World
8:         Lock given memory region in cache
9:         Enable Interrupts
```

Overall, the entire process is implemented as roughly 200 lines of C and assembly code in the Secure World kernel which is a minimal increase in the Trusted Code Base (TCB) of the Secure World.

Genode utilizes the ARM Virtualization Extensions for the TZ-VMM scenario. [21] Therefore, to establish World-Shared Memory, the Genode bootstrap process is modified to map the RAM region that is later allocated to the Normal World VM to a designated virtual address space in the Secure World. Once defined, the mapping can be added in the Platform constructor of the bootstrap process to create the page table entries in the Secure World for the Normal World RAM space, effectively creating

World-Shared Memory. The entries are created with the necessary flags, among them the NS bit that indicates the virtual addresses resolve to non-secure physical space.

Once we deploy CacheLight, we are able to detect whenever the CacheKit module is attempting to lock malicious code into the cache. By doing security checks on the PA being locked into cache, CacheLight can detect anomalous behavior and if needed deploy inspection tools. If the attempt to lock the cache is malicious, it can then flush the caches and run memory introspection tools to determine the nature of the attack and retrieve any relevant data for forensic analysis. On the other hand, if the request is determined to be legitimate, CacheLight can service it by taking advantage of world-shared memory.

Chapter 6

RELATED WORK

Overtime, rootkits and the methods for detecting and defending against them have been in an evolutionary race of hide and seek. Originally, persistent rootkits needed to modify nonvolatile storage to survive system power cycles which meant that file integrity checking tools could effectively detect them. [22] Therefore, rootkits switched to reside only in the operating system kernel memory to defeat this storage-based detection. To detect this new type of rootkit, defenders acquire the system memory using a dedicated secure coprocessor [23] or physical hardware [24]. In their search to acquire higher root privileges, attackers have developed several different rootkits. Virtual machine based rootkits (VMBR) insert a customized malicious hypervisor beneath the currently running operating system. [25] Firmware based rootkits infect the firmware on I/O devices [26] or the system BIOS. [27]

As their counterpart, new hardware and software rootkit detectors with higher privilege are also proposed. Hypervisors are commonly used to introspect the untrusted operating system [28]. However, vulnerabilities are frequently found in the hypervisors. This gave rise to the use of hardware features, such as security extensions in various processors. In this paper, we cover ARM TrustZone, however, each developer has their own set of hardware security extensions. For example, AMD has SVM [29] and Intel provides TXT [30]. These hardware features provide a trusted execution environment (TEE) with guaranteed isolation. The TEE provides an environment with the highest privilege that defenders can claim as their own as seen with TrustZone. CacheKit seeks to exploit weaknesses in these new hardware defense features to evade detection by the OS kernel. Another proposed design that has much the same goals is Shadow Walker, as it exploits the I-TLB and D-TLB coherency problem in the Intel architecture to hide the rootkits. [31] There is also an implementation called Cloaker that was used as one of the comparisons in Table 6.1 for CacheKits efficiency. Cloaker can hide its presence by locking the page translation it altered in the translation look-aside buffer. [32] However, we note that CacheKit provides a new level of stealth as it is able to evade physical memory inspection by hiding in the Normal World cache.

Table 6.1: Comparisons Between CacheKit and Other Attacks with Respect to Various Existing Detection Methods [1]

Detection Methods	User	Kernel	VMBR	SMMR	Cloaker	CacheKit
App level detection	YES	NO	NO	YES	NO	NO
OS level detection	YES	NO	NO	NO	NO	NO
VMM level detection	YES	YES	YES	NO	NO	NO
Coprocessor Based	YES	YES	YES	NO	NO	NO
TEE Based	YES	YES	YES	NO	NO	NO
Physical Memory Check	YES	YES	YES	YES	YES	NO

CacheKit is the only one able to evade the physical memory check. Furthermore, by mapping the cache lines to unused I/O space we ensure that the malicious code cannot be examined. In the worst case, it is evicted and then destroyed before it can be inspected and analyzed. This prototype is implemented on the i.MX53 development board and thus the immediate and complete application of the concept is limited to similar single Cortex-A8 processor systems. While this is only a subset of all ARM processors, it still poses a major threat as a nearly undetectable rootkit mechanism in embedded systems. To the best of our knowledge, there have been no defense mechanisms proposed for this kind of attack.

Chapter 7

FUTURE WORK

While this initial implementation of CacheLight shows great potential, there is still much work and refinement to be done. Most notably is more quantitative performance analysis of CacheLight. We need to collect data to determine the exact performance impact of CacheLight on the system based on the size of the data being loaded and the amount of cache to be locked, similar to the analysis performed in CacheKit. Further implementation independent testing is also necessary to ascertain and prove the process for tagging entries in the cache with the NS bit. Additionally, there are further applications and checks for which CacheKit could be leveraged to tighten security. For example, limiting or regulating how much of the cache a request can lock.

Furthermore, there is the case where a rootkit is loaded, but not locked, in the cache with a remapped address so the attempt fails. However, this attempt would not be retrievable for analysis, therefore, in such cases valuable forensic data is lost about the attack and attacker. Perhaps there could be work done to expand the robustness of the approach in such cases.

Finally, there is still more work that can be done on the response after an attack is detected and verified. Looking deeper into the what inspection tools to run in case CacheLight flags potential malicious code and the appropriate response mechanisms to employ if the tool does find malicious code in memory.

Chapter 8

CONCLUSION

In this paper we present CacheLight, a lightweight approach to preventing malicious use of cache locking mechanisms while allowing time-critical applications to legitimately utilize them to ensure execution times in embedded and real-time systems. In recent years, we have seen many advances by industry and academia to increase the security of computer systems. Namely, we focus on TrustZone, a hardware-based security extension developed by ARM to secure their processors which power a vast majority of embedded systems. Evidently, just as security mechanisms have evolved, so have the attacks and attackers that seek to circumvent them. CacheKit is a novel approach that exploits the cache locking mechanisms offered on some ARM processors to evade memory introspection from tools running in the Secure World of the processor. This gives the rootkit a new level of unprecedented stealth that would leave millions of devices using these processors severely vulnerable. By hiding in the Normal World cache of the processor, the new rootkit makes itself undetectable by any of the defense mechanisms available today, including memory inspection. Furthermore, by exploiting the virtualization extensions and modifying the physical address that it maps to, it renders any possibility of malicious code recovery for forensic purposes impossible. Therefore, we propose a novel solution that leverages the TrustZone extensions to prevent such an attack while still giving applications access to the timing facilities offered by the processor.

The simplest solution described is to not allow application in the Normal World to perform cache locking operations. However, as discussed, these are features that are critical to the performance and timing requirements of real-time and many embedded systems. Therefore, disabling their use is not practical. After a detailed exploration of the ARM architecture and attack features, we propose CacheLight which allows the Normal World to perform cache locking through requesting it as a service from the Secure World. All that is needed is the addition of a Secure Monitor Call to the Secure World that will request it to lock a given address to a given cache way. With a minimal increase in the Trusted Code Base to handle this SMC, the operating system running in the TEE can then validate and verify the validity of this request and prevent any malicious code from being hidden in the cache.

To verify the functionality of the defense approach, the Genode Operating System framework is deployed on the i.MX53 development board running the TZ-VMM scenario to boot Linux in the normal world. The Linux kernel is modified to handle four SMCs, to which we add our own call for the request to lock a certain cache way. The address, cache way, and size of memory to load are passed as arguments to the call. Upon world switch, the Secure World can now handle and verify the validity of this call as it has all the necessary information. The VA to PA translation registers are used to resolve the given virtual address to a physical address in system memory. This address can then be verified to ensure actual data is being used. More importantly, it ensures that there is no incoherence between the cache and RAM. That is, whatever is in the given address in RAM will be stored and locked in the cache, thereby ensuring that any data that will persist in the cache not only maps to a valid address in memory but is also consistent with what was loaded to that memory region, effectively bringing the contents of the cache to light. Therefore, the memory introspection tools in the Secure World do not have to worry about not being able to scan Normal World cache because we have ensured that the data is also present in memory. Additionally, because the Secure World does not hand control back to Normal World after verifying the address, but rather performs the loading and locking on behalf of the Normal World, the attacker cannot bypass the security checks by passing different addresses in the arguments. The remapping of a virtual address to a physical address in unused I/O memory provides a very strong attack signature. Therefore, by doing security checks on the PA being locked into cache, CacheLight can detect any anomalous behavior and if needed deploy inspection tools.

Should CacheLight find that the attempt to lock the cache is malicious, it can then flush the caches and run memory introspection tools to determine the nature of the attack and retrieve any relevant data for forensic analysis. On the other hand, if the request is determined to be legitimate, CacheLight can service it by taking advantage of world-shared memory. By creating the necessary translation tables, Secure World can access Normal World memory with an NS bit of 1 and therefore, the loading will tag the cache entries with NS = 1. This way, when Normal World attempts to use the locked entires, the TLB look-up will trigger a cache hit and the data will be readily accessible as if the Normal World had made the entry.

Therefore, CacheLight can successfully prevent malicious code from hiding from SW introspection tools in the NW cache for any significant amount of time. Note that a compromised NW kernel would still be able to load malicious code into the cache, however, recall that one of the essential requirements for this attack to work is that the rootkit must remain persistent in cache long enough for it to be of use. This cannot be achieved without the hardware-level cache locking mechanisms provided by the processor. Therefore, any rootkit installed in the cache would be evicted from the cache thereby rendering it useless.

In conclusion, CacheLight provides a lightweight and novel approach to preventing a new generation of attacks to exploit the cache locking mechanisms provided by processors. Additionally, while we present a solution for the ARM architecture, the approach can be generalized to any architecture that employs the same execution separation idea. If the attack can be modified to a new architecture, then so can the defense. Moreover, CacheLight incurs the overhead of a world-switch for the set-up of the time-critical data. However, the initial setup of locking data in the cache is already expected to be expensive so that the performance and timing requirements can be met once the setup is done and the application running. CacheLight makes additional overhead to the setup process but not the execution of the time-critical process that requested the lock. Given that it provides security against an otherwise undetectable attack, the trade-off in setup time is extremely worthwhile.

REFERENCES

[1] N. Zhang, H. Sun, K. Sun, W. Lou, and Y. T. Hou, "CacheKit: Evading memory introspection using cache incoherence," in *Proceedings of the 1st IEEE European Symposium on Security and Privacy.* Saarbrcken, GERMANY: IEEE, 2016.

[2] ARM, "ARM Cortex-A8 Technical Reference Manual," http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/DDI0344K_cortex_a8_r3p2_trm.pdf, 2006-2010.

[3] (2016) Arm: Media fact sheet. ARM News. [Online]. Available: https://www.arm.com/-/media/arm-com/news/ARM-media-fact-sheet-2016.pdf?la=en

[4] F. Zhang, J. Wang, K. Sun, and A. Stavrou, "Hypercheck: A hardware-assistedintegrity monitor," vol. 11, no. 4, July 2014, pp. 332–344.

[5] 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 21st ACM Conference on Computer and Communications Security (CCS).* Scottsdale, Arizona: ACM, Nov. 2014, pp. 90–102.

[6] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia, "Trustdump: Reliable memory acquisition on smartphones," in *Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS).* Wroclaw, Poland: Springer, Sep. 2014, pp. 202–218.

[7] Y. Lu, L. Lo, G. R. Watson, and R. G. Minnich, "Car: Using cache as ram in linuxbios." http://rere.qmqm.pl/mirq/cacheasramlb09142006.pdf, 2006.

[8] V. J. Zimmer, M. A. Rothman, and S. M. Datta, "Using a processor cache as ram during platform initialization," US Patent 20,040,103,272.

[9] ARM, "ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html, 2012.

[10] T. Alves and D. Felton, "Trustzone: Integrated hardware and software security," https://github.com/laginimaineb/cve-2016-2431, 2004.

[11] ARM, "ARM Cortex-A Series Programmerś Guide for ARMv8-A," http://infocenter.arm.com/help/topic/com.arm.doc.den0024a/index.html, 2015.

[12] ——, "ARM Security Technology Building a Secure System using TrustZone Technology," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html, 2009.

[13] T. Liu, M. Li, and C. J. Xue, "Instruction cache locking for embedded systems using probability profile," *Journal of Signal Processing Systems*, vol. 69, no. 2, pp. 173–188, Nov 2012.

[14] Y. Liang, T. Mitra, and L. Ju, "Instruction cache locking using temporal reuse profile," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 9, pp. 1387–1400, Sept 2015.

[15] S. Mittal, "A survey of techniques for cache locking," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 21, no. 3, pp. 49:1–49:24, May 2016. [Online]. Available: http://doi.acm.org.ezproxy1.lib.asu.edu/10.1145/2858792

[16] K. Anand and R. Barua, "Instruction-cache locking for improving embedded systems performance," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 3, pp. 53:1–53:25, Apr. 2015. [Online]. Available: http://doi.acm.org.ezproxy1.lib.asu.edu/10.1145/2700100

[17] Genode-Labs, "Genode: Operating system framework," https://github.com/genodelabs/genode, 2017.

[18] N. Fenske, *Genode Operating System Framework Foundations.* Genode Labs, 2017.

[19] (2017) An exploration of arm trustzone technology. Genode OS Documentation and Articles. [Online]. Available: https://genode.org/documentation/articles/trustzone

[20] ARM, "SMC CALLING CONVENTION System Software on ARM Platforms," http://infocenter.arm.com/help/topic/com.arm.doc.den0028b/ARM_DEN0028B_SMC_Calling_Convention.pdf, 2016.

[21] (2017) An in-depth look into the arm virtualization extensions. Genode OS Documentation and Articles. [Online]. Available: https://genode.org/documentation/articles/arm_virtualization

[22] 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 (CCS).* Fairfax, VA: ACM, Nov. 1994, pp. 18–29.

[23] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot - a coprocessor-based kernel runtime integrity monitor," in *Proceedings of the 13th USENIX Security Symposium (Security).* San Diego, CA: USENIX, Aug. 2004, pp. 179–194.

[24] E. Chan, S. Venkataraman, F. David, A. Chaugule, and R. Campbell, "Forenscope: A framework for live forensics," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC).* ACM, 2010, pp. 307–316.

[25] J. Rutkowska, "Subverting vistatm kernel for fun and profit," Black Hat Briefings, 2006.

[26] J. Heasman, "Implementing and detecting a pci rootkit," *NGSSoftware Insight Security Research.*

[27] S. Embleton, S. Sparks, and C. Zou, "Smm rootkit: a new breed of os independent malware," Security and Communication Networks, 2013.

[28] A. M. Azab, P. Ning, and E. C. Sezer, "A hypervisorbased integrity measurement agent," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC).* ACM, 2009, pp. 461–470.

[29] *Amd64 Architecture Programmers Manual.*

[30] *Intel Software Development Guide.*

[31] S. Sparks and J. Butler, "Shadow walker: Raising the bar for rootkit detection," *Black Hat Japan*, pp. 504–533.

[32] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "Cloaker: Hardware supported rootkit concealment," in *Proceedings of the 29th IEEE Symposium on Security and Privacy (Oakland).* Oakland, CA: IEEE, May 2008, pp. 296–310.