Attack and Defense with Hardware-Aided Security

Ning Zhang

Dissertation submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Application

Wenjing Lou, Chair
Ing-Ray Chen
Yiwei T. Hou
Elaine R. Shi
Kun Sun
Danfeng Yao

July 22, 2016
Falls Church, Virginia

Keywords: Trusted Execution Environment, Hardware-Assisted Security, Secure Execution, Cache, Rootkit

Attack and Defense with Hardware-Aided Security

Ning Zhang

ABSTRACT

Riding on recent advances in computing and networking, our society is now experiencing the evolution into the age of information. While the development of these technologies brings great value to our daily life, the lucrative reward from cyber-crimes has also attracted criminals. Unwanted software is delivered to the victims through various methods such as remote exploitation and social engineering. As computing continues to play an increasing role in the society, security has become a pressing issue. Failures in computing systems could result in loss of infrastructure or human life, as demonstrated in both academic research and production environment. With the continuing widespread of malicious software and new vulnerabilities revealing every day, protecting the heterogeneous computing systems across the Internet has become a daunting task.

Our approach to this challenge consists of two directions. The first direction aims to gain a better understanding of the inner working of both attacks and defenses in the cyber environment. It was said in Art of War by Sun Tzu that if you know your enemies and know yourself, you will not be imperiled in a hundred battles. I share the same belief that in order to design systems that are resistant to cyber attackers, it is necessary to understand how cyber-attack works. Under this direction, part of our research focuses on examining the evolution of cyber-attack. Meanwhile, our other direction is designing secure execution environments using security features offered by hardware. Hardware-aided security offers the capability to bootstrap a secure environment even when the software environment is compromised. Under this direction, our research focuses on providing trusted environment in an adversarial setting with the presence of powerful attackers.

The next generation of computing consists of powerful elastic cloud and mobile devices inter-connecting via fast and reliable networks. Therefore to provide a complete solution, our work spans both the cloud and mobile endpoint.

Attack and Defense with Hardware-Aided Security

Ning Zhang

GENERAL AUDIENCE ABSTRACT

Riding on recent advances in computing and networking, our society is now experiencing the evolution into the age of information. While the development of these technologies brings great value to our daily life, the lucrative reward from cyber-crimes has also attracted criminals. As computing continues to play an increasing role in the society, security has become a pressing issue. Failures in computing systems could result in loss of infrastructure or human life, as demonstrated in both academic research and production environment. With the continuing widespread of malicious software and new vulnerabilities revealing every day, protecting the heterogeneous computing systems across the Internet has become a daunting task. Our approach to this challenge consists of two directions. The first direction aims to gain a better understanding of the inner working of both attacks and defenses in the cyber environment. Meanwhile, our other direction is designing secure systems in adversarial environment.

Dedication

To my mother

Acknowledgments

I have had countless mistakes and failures in life. Without the support and encouragement from my teachers, colleagues, family, and friends, I would not have become what I am today.

First and foremost, I would like to express the deepest gratitude to my advisor Dr. Wenjing Lou. She supports me to pursue my research interest, always encourages me to explore the unknown and dream big. Her rigorous research methods inspire me to seek for the truth in science. I am also thankful for the career guidance she offered me.

I would also like to thank Dr. Kun Sun. He has been a mentor for me since my early days in the Ph.D. career. He has shared his rich experiences as a security researcher and guided me through many details. I sincerely thank him for being an invaluable friend and mentor.

I would also like to thank my committee members Dr. Elaine Shi, Dr. Dephane Yao, Dr. Ing-Ray Chen, Dr. Tom Hou for their insightful comments and suggestions on my research.

I thank Dr. Wenhai Sun, Ethan Gabriel, Dr. Xuxian Jiang, Ruide Zhang, Dr. Ming Li, Tao Jiang and Dr. He Sun for their active participation and helpful discussions in our research works.

I would also like to thank the generous support from Raytheon company in supporting my Ph.D. study. I thank Susan Battaller for encouraging me to apply for the fellowship.

I also like to thank my colleagues at the complex network and security research (CNSR) laboratory at Virginia Tech, Yao Zheng, Dr. Bing Wang, Dr. Qiben Yan, Changlai Du, Dr. Xinghua Li for the discussions.

Lastly, I would like to thank my family, especially my mother. Though passing away early in life, my mom had cared and raised me as a single mother. Without her love, I would never have been able to stand strong through the numerous ups and downs in life. I also would like to thank my wife, Bei Li, for all her support during my study in the Ph.D. career. I thank my uncle for showing me the world of computing at an early age.

Funding Acknowledgments

Research work presented in this dissertation is supported in part by National Science Foundation (NSF) under grants CNS-1217889, CNS-1446478, CNS-1405747 and CNS-1443889. Ning Zhang is also supported by Raytheon company under Raytheon Advance Scholar.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation - Attack and Defense in Modern World Computing

Harboring the great benefits of modern day computer networks, our society is becoming more and more tightly coupled with information systems. Compromised computing system could lead to loss of infrastructure [130, 88] or human lives [103]. With the increasing number of malicious software and new vulnerabilities revealing everyday [118], protecting the computing systems across the Internet has become a pressing issue. I follow two directions to tackle this challenge, understanding advanced attack and building secure system in adverse environment.

- The first half of my study aims to gain better understand the evolution of attacks in the cyber environment. It is said that if you know your enemies and know yourself, you will not be imperiled in a hundred battles [183]. I share the same belief that in order to design future systems that are resilient to cyber attackers, we have to understand the evolution of cyber attacks. When advanced persistent threat (APT) is so prevalent, one of the challenges is how to assess the security state of the computing system. System checking via memory inspection is a popular method to check the integrity of software running on the system. In our research, we are interested in the architectural foundation of system introspection.
- The second part of my study tackles the challenge of providing trusted execution environment in adversarial environments. With the ever growing complexity in software system nowadays, vulnerabilities are discovered and disclosed daily. Despite decades of research and development in programming language and operating systems, secure system remains an active research area. Hardware-aided security offers a unique advantage to bootstrap a trusted environment even when even when the system software is compromised. Following this direction, our research focuses on design and implementation of secure execution environments for both Cloud and mobile endpoints in adverse setting.

1.2 Research Contribution in Understanding Advance Attack

The first half of my dissertation focuses on understanding the evolution of the attacks. More specially, advance stealthy rootkits in the most widely used micro-architectures, x86 and ARM, are studied.

1.2.1 Exploiting Physical Address Space Inconsistency

The first work aims to understand the inner working of memory forensic, and what the architectural assumptions of this process. With the growing complexity of computing systems, memory based forensic techniques are becoming instrumental in digital investigations. Digital forensic examiners can unravel what happened on a system by acquiring and inspecting in-memory data. Meanwhile, attackers have developed numerous anti-forensic mechanisms to defeat existing memory forensic techniques by manipulation of system software such as OS kernel. To counter anti-forensic techniques, some recent researches suggest that memory acquisition process can be trusted if the acquisition module has not been tampered with and all the operations are performed without relying on any untrusted software including the operating system. However, in this work, we show that it is possible for malware to bypass the current state-of-art trusted memory acquisition module by manipulating the physical address space layout, which is shared between physical memory and I/O devices on x86 platforms. This fundamental design on x86 platform enables an attacker to build an OS agnostic anti-forensic system. Base on this finding, we propose Hidden in I/O Space (HIveS) which manipulates CPU registers to alter such physical address layout. The system uses a novel *I/O Shadowing* technique to lock a memory region named *HIveS memory* into I/O address space, so all operation requests to the HIveS memory will be redirected to the I/O bus instead of the memory controller. To access the HIveS memory, the attacker unlocks the memory by mapping it back into the memory address space. Two novel techniques, *Blackbox Write* and *TLB Camouflage*, are developed to further protect the unlocked HIveS memory against memory forensics while allowing attackers to access it. A HIveS prototype is built and tested against a set of memory acquisition tools for both Windows and Linux running on x86 platform. Lastly, we propose potential countermeasures to detect and mitigate HIveS.

1.2.2 Exploiting Cache Incoherence

For the second work in the attack domain, we turn our attention to the mobile devices. With mobile phones becoming an increasingly pervasive computing platform, new attacks targeting on compromising the mobile OS are emerging. ARM processors, powering over 90% mobile phones, introduced 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 the TrustZone technology, a number of memory integrity checking schemes have been proposed in the secure world to detect malicious memory modification of the normal world. Some of these

schemes have been integrated in commercial products. In this work, we develop a new rootkit called *CacheKit* that can bypass the TrustZone-based memory integrity checking mechanism. The basic idea is based on our observation and verification of a TrustZone cache coherency problem - code in one TrustZone world cannot access the cache content in the other world. In other words, though the secure world has the privilege to access the *memory* of the normal world, it does not have access to the *cache* content of that world. *Cache-as-RAM* is used in Cachekit to enable code execution in cache, and two newly developed techniques *cache locking* and *physical address space manipulation* are used to lock and conceal the malicious code in the cache of the normal world. CacheKit leaves no footprint in memory and can successfully evade introspection from TrustZone. We implemented CacheKit on Freescale i.MX53 development board with a 1 GHz Cortex-A8 processor. We also discuss potential countermeasures to mitigate this newly discovered attack.

1.3 Research Contribution in Trusted Execution Environment in Adverse Environment

The second half of my dissertation focuses on providing a trusted execution environment in the presence of powerful adversary capable of gaining the control of the software system.

1.3.1 Trusted Data-Intensive Execution in Untrusted Cloud Computing

The first work on trusted computing focus on securing the workloads in cloud computing. The security and privacy of user data has become a major concern in the cloud computing era. Cryptographic solutions based on secure computation outsourcing have been extensively studied in order to protect the security and privacy of user data. However, these solutions either suffer from forbiddingly high computation overhead or are only applicable to certain special classes of computations. In this work, we tackle the challenge of secure computation outsourcing using an entirely different approach - the idea is to have a secure execution environment in the cloud such that user data can be processed in plain text format without compromising its confidentiality. We propose a TrUsted Data-intensive ExeCution (TUDEC) environment optimized for data applications in the cloud. TUDEC is a new system architecture, designed to provide a secure environment for arbitrary data computations in the cloud server. Using a very small trusted computing base including only firmware and hardware, TUDEC is able to provide user VM with isolation against both the legacy host and neighboring VMs. Such isolation is unique in that it provides *protection against any software-based attacks*. By *direct interrupt delivery*, *interrupt rerouting* and *IOMMU configuration lock*, TUDEC enables close to bare metal computation and I/O performance without sacrificing any security guaranteed. We built a prototype and showed the high efficiency of TUDEC. In particular, when the server is heavily loaded, the TCP bandwidth of the guest VM in TUDEC is significantly better than the current state of art secure execution environment design.

1.3.2 Cache-Assisted Secure Execution on ARM Processors

The second work on trusted computing focuses on the mobile platform. Recognizing the pressing demands to secure mobile applications, ARM TrustZone has been adopted in both academic research and commercial products to protect sensitive code and data in a privileged isolated execution environment. However, the design of TrustZone cannot prevent physical memory disclosure attacks such as cold boot attack from gaining unrestricted read access to the sensitive contents in the Dynamic Random Access Memory (DRAM). A number of Soc-bound execution solutions have been proposed to defeat the cold boot attacks by constraining the secrets only in CPU registers, CPU cache, or internal RAM. However, when the mobile OS, which is responsible for creating and maintaining the SoC-bound execution environment, is compromised, all the secrets will be leaked out. In this work, we design and develop a cache-assisted secure execution mechanism called CaSE on ARM processors to defend against sophisticated attackers who can launch multivector attacks including software attacks and hardware memory disclosure attacks. CaSE utilizes TrustZone to create a cache based isolated execution environment, which can protect both sensitive code and data against the malicious mobile OS and the cold boot attacks. Since the encrypted secure application is loaded into the CPU cache and then decrypted within the processor, we can protect not only the sensitive data but also the code confidentiality against cold boot attacks. Also, we use the memory separation and the cache separation mechanism provided by TrustZone to protect the cached application against malicious mobile OS. We implement a prototype of CaSE on the i.MX53 running ARM Cortex A8 processor. The experimental results show that CaSE incurs small impacts on system performance when executing cryptographic algorithms including AES, RSA, and SHA1.

1.4 Document Organization

The rest of the dissertation is organized as follows. Chapter 2 and Chapter 3 focus on presenting the first half of the research in the dissertation, where efforts are on understanding advance memory attacks. In Chapter 4 and Chapter 5, the second half of the dissertation, which focuses on designing secure system in adverse environment, is presented. Chapter 6 summarizes the dissertation and discusses future work.

Chapter 2

Attack - HIveS: Exploiting Physical Address Space Inconsistency against Physical Memory Forensic

2.1 Introduction

Digital forensics is the science on collecting and presenting digital evidence. With the ever increasing use of computing systems in our daily life, computers and networks have become not only the personal portal to instant information, but also a platform that criminals exploit to commit crimes. Digital forensics is now one of the services sought at the very beginning of all types of investigation - criminal, civil, and corporate [65, 79].

Disk forensic methods and tools have matured in the past two decades, offering comprehensive capabilities to extract and visualize artifacts from nonvolatile storage images. With the prevalence of memory hiding techniques and the need to evade disk forensic, adversaries are starting to hide the presence of malicious code and data only in the memory [66, 95, 91]. To tackle this problem, forensic examiners are increasingly relying on live memory forensics to uncover the malicious contents in the memory [65].

There are two general memory acquisition approaches: *software based* approach and *hardware based* approach. Software based solutions rely on a trusted memory acquisition module in the operating system to acquire the memory through the processor [175, 133]. Hardware based solutions often utilize dedicated I/O devices, such as network interface card, to capture physical memory image via direct memory access (DMA) [69, 158, 194] with the processor totally bypassed. Some hardware based approaches use the remanence of physical memory to extract sensitive data from memory module in systems that are powered off for a short time [70, 111].

To counter live memory forensics, attackers have developed a number of anti-forensic techniques

to sabotage the memory acquisition process [112]. Current anti-forensic techniques against software based memory acquisition rely on manipulating the software used in the memory acquisition process. Some examples include modifying the acquisition module or the OS kernel [113, 172, 91], hooking operating system APIs [168], or installing a thin hypervisor on the fly [162]. Based on this observation, Stüttgen et al. recently suggested that the memory acquisition process can be trusted with two conditions. The first one is that the acquisition module has not been tampered with, and the second one states all the operations are performed without relying on the operating system or any other untrusted software [175]. However, in this work, we show that this assumption is not true by presenting Hidden in I/O Space (HIveS), an operating system (OS) agnostic anti-forensic mechanism, that is capable of evading the most updated software based memory forensics tools.

Physical address space on x86 platform is shared between physical memory and I/O devices. Memory access to a physical address gets directed to either the memory controller or the I/O bus based on where it is located in the address space layout. This physical address layout is also what memory forensics tools use to understand where the physical memory regions are located. Memory forensic tools obtain this layout information by interacting with operating system or BIOS, and they assume this layout is correct and updated. We show that this condition can be easily violated by presenting HIveS. HIveS alters the machine physical address layout while the system is in operational state by modifying registers in the processor. With this mispresented address layout, HIveS can conceal a memory region called *HIveS memory* from being observed and acquired by memory forensics tools.

The basic idea is to map (or lock) the HIveS memory into the I/O space, so that any operation on the physical memory address will be redirected to the I/O bus instead of the memory controller. When the HIveS memory is locked, its memory contents cannot be accessed by any processor, including the one(s) controlled by the attacker. When the attacker wants to access the HIveS memory, she would first unlock the memory region by mapping it back into the memory address space. To protect the unlocked HIveS memory against memory forensics, we propose two novel techniques, *Blackbox Write* and *TLB Camouflage*. Blackbox Write enables only write access to the HIveS memory by creating asymmetric read and write destinations between the memory space and the I/O space. TLB Camouflage exploits TLB cache incoherency among multi-core processors to ensure exclusive read and write access for a single processor core to the HIveS memory.

HIveS is operating system agnostic, since it only changes the system hardware configurations. We build a prototype of HIveS on an x86 desktop with an AMD FX processor running both Windows and Linux. Since HIveS conceals the presence of malware without changing any system software including BIOS, hypervisor or OS kernel, it can effectively defeat the most updated software based memory acquisition tools on both Windows and Linux. Furthermore, we extend HIveS with a number of existing anti-forensic techniques, such as *RAM-less encryption* and *Cache based I/O storage*, to defeat hardware based memory acquisition approaches.

We propose several countermeasures for detecting and mitigating HIveS. One seemingly simple solution is to directly inspect the CPU registers that may have been manipulated by HIveS. However, since legitimate peripheral device drivers may also change the same set of CPU registers,

it remains a challenge to distinguish normal configurations from malicious usages, and maybe impossible without crashing the system on some hardware platforms.

To summarize, we make the following contributions.

- We present HIveS, a system that exploits hardware features in x86 platform to subvert the foundation of memory acquisition. HIveS is an OS agnostic anti-forensic mechanism that can defeat memory forensic techniques by concealing the HIveS memory in the I/O space.
- We develop two novel techniques to enable covert operations on the unlocked HIveS memory against memory forensics. Blackbox Write grants only the write privilege to the HIveS memory, and TLB Camouflage can grant malicious users exclusive read and write access to the HIveS memory.
- A prototype of HIveS is built on the x86 platform to demonstrate its capability on concealing the HIveS memory against a number of most updated memory forensics tools on both Windows and Linux.
- We propose potential countermeasures to detect and mitigate HIveS. As an arms race, we show that HIveS can be enhanced to further evade hardware based memory acquisition solutions.

The remainder of the chapter is organized as follows. Section 2.2 describes some background knowledge on x86 memory address space. We present the HIveS framework in Section 2.3 and discuss its extensions in Section 2.4. A prototype implementation is detailed in Section 2.5. We propose potential countermeasures in Section 2.6. In Section 2.7, we present the future work of HIveS. Section 2.8 discusses the related works. Finally, we conclude the chapter with Section 2.9.

2.2 Background

The entire range of memory addresses accessible by x86 processors is often referred to as *physical address space*. Contrary to popular believes, the length of such address space usually does not equal to the amount of actual physical memory installed on the platform. This is because some of the address is mapped to the bus for I/O devices, instead of dynamic random access memory (DRAM). A typical memory layout of systems with AMD processors is shown in Figure 2.1, where the shaded areas are backed by DRAM, and the areas without shade are backed by I/O devices. 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).

The memory setting of an x86 system is initialized by the BIOS at hardware reset and parsed by the operating system during the system bootstrap [68]. The layout is configured via several configuration registers in the North Bridge (NB) and the processor. DRAM Base/Limit register

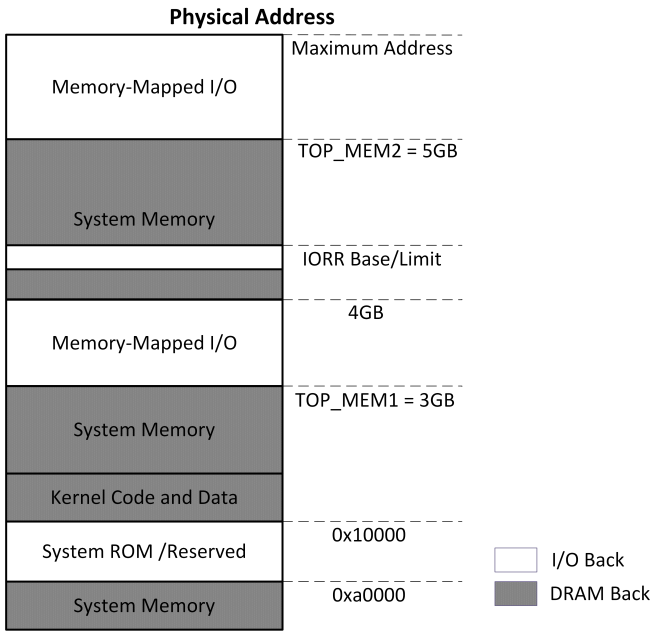

Figure 2.1: Physical Address Layout on AMD Architecture

pair is among the earliest ones configured by the BIOS. They define the ranges of physical address space mapped to DRAM in the north bridge. Any access to these areas will be forwarded to the DRAM Controller (DCT). These registers are configured by the BIOS with the result of system memory probing during hardware initialization. Therefore they are designed to be lock-once (i.e., write-once). The values cannot be changed until the next system reset.

The next set of registers that shapes the memory layout consists of two Mode Specific Registers (MSR) Top Of Memory (TOM) registers. AMD processors allow system software to use TOM registers to specify where memory accesses are directed for a given address range [38]. There are two TOM registers, *TOP_MEM1 (TOM1)* and *TOP_MEM2 (TOM2)*. Figure 2.1 shows that the address range from 0 to TOM1 as well as the address range from 4GB to TOM2 are set as system memory on this AMD system. Access requests within these two ranges are directed to the DRAM, while requests outside these two ranges are directed to the I/O space. The purpose of these two registers is to offer the operating system software the ability to carve out large memory space to organize DRAM and I/O devices. Even though they can be changed even when the system is operational, unlike the DRAM Base/Limit register, it is rare to change the memory address allocation after the system starts up. This is because the DRAM boundaries, governed by DRAM Base/Limit registers, have already been determined. Lastly, systems usually stop functioning if these registers are changed, since the OS kernel was not expecting the change of hardware configuration while the system is running.

The last set of registers that shapes the layout is also MSR in the processor. They are the Input Output Remap Registers (IORR). These set of registers can create a special mapping beyond the

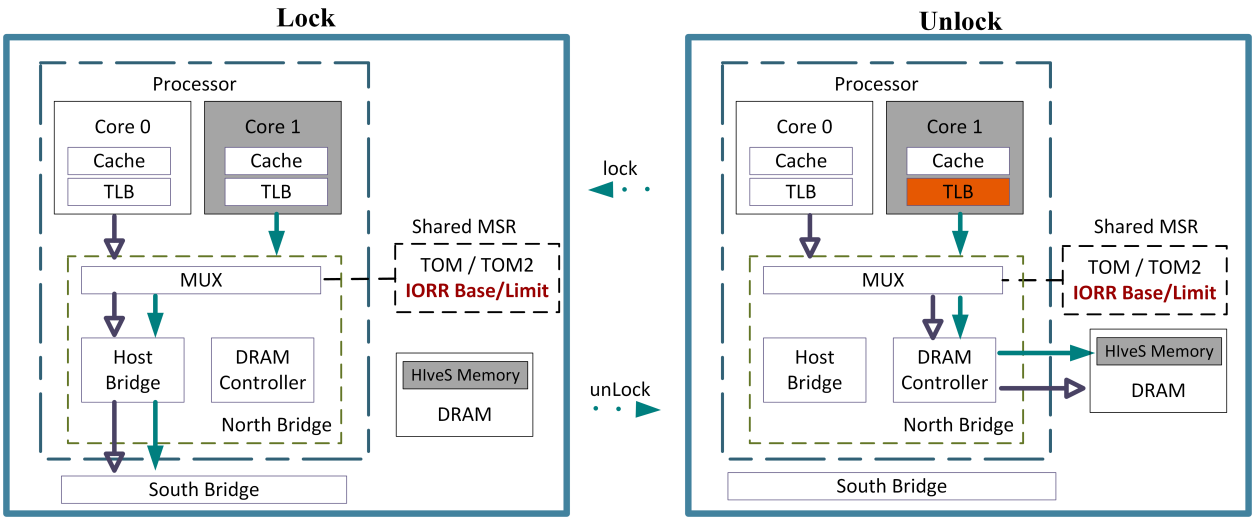

Figure 2.2: Architecture of HIveS

base setting to direct specific read/write access of any address space between the I/O space and the DRAM space. This set of registers are designed to enable system software to shadow ROM device in memory to improve the system performance.

2.3 HIveS Framework

In the ongoing battle between attackers and digital forensics examiners, memory acquisition is becoming an important technique for evidence collection. From the perspective of an attacker, we design HIveS, an anti-forensic system. It is capable of evading acquisition by software based memory forensics tools on a designated range of physical memory chosen by the attacker. We call this range of memory *HIveS memory*. It can be used by attackers to store malicious code or sensitive data.

A high level block diagram of the HIveS system is shown in Figure 2.2. For simplicity, we show a generic x86 multi-core architecture with one processor consisting of two cores. Each processor core has its own cache and TLB.

When a processor core needs to access the DRAM memory, it sends a request to the north bridge. The MUX inside the north bridge is responsible of forwarding the memory request to either the DRAM controller or the south bridge based on the physical address layout. This layout was initialized by the BIOS, then further defined by the operating system using model-specific registers (MSRs) including the top of memory (TOM) registers and the I/O range registers (IORRs). When the physical address is mapped to the I/O space, the request is forwarded to the south bridge. When the physical address falls in the DRAM range, the memory request goes through the DRAM controller to the physical memory.

HIveS has two states, *locked* and *unlocked*. When it is in the locked state, the HIveS memory is completely inaccessible to any processor core. This is because all access attempts are forwarded to the I/O space once HIveS is locked. While the system remains in this state, even the malicious core (e.g., Core 1 in Figure 2.2) cannot access the HIveS memory. When the attacker needs to access the HIveS memory, she can set HIveS to unlocked state, where only the malicious core can access the HIveS memory, and memory requests from all other cores are redirected to another DRAM region. Lastly, since HIveS relies only on hardware configurations to conceal the HIveS memory, it is OS agnostic. Moreover, it leaves no trace in memory. Unlike some of the current rootkits that modifies kernel data structures or operating system APIs, HIveS cannot be detected by checking the integrity of the OS.

2.3.1 Inaccessibility in the Locked State

Considering the use case of a password stealing rootkit, whose goal is to steal passwords and store them quietly in some place before an opportunity to exfiltrate, there is no need for the rootkit to read from or write to the memory where the stolen passwords are stored until it is ready to transmit. HIveS is designed to an anti-forensic tool, so we develop a novel *I/O Shadowing* technique to block all processor cores from accessing the HIveS memory. The basic idea of I/O shadowing is to dynamically manipulate the configuration of a memory range so that even if it is backed by the DRAM in the physical address space, any read/write request will be redirected to the I/O space. The real contents in the DRAM memory are shadowed by the memory-mapped I/O (MMIO).

Among the various controls that shapes the memory layout, there are two MSRs that can be controlled by the system software when the system is operational. They are TOM and IORR.

Though TOM registers can be modified after the system boots up, any modification of the TOM registers can greatly affect the system stability, since the OS kernel uses the TOM registers in many default system settings. Furthermore, TOM modifications can only change the boundary between the default I/O area and the DRAM area. Even if system instability was not an issue, the manipulation would be very limited.

We instead use I/O range registers (IORRs) to adaptively prevent all processor cores from accessing the HIveS memory. IORRs are variable-range memory type range registers (MTRRs). They can be used to specify if reads and writes in any physical address range should map to system memory or memory-mapped I/O (MMIO). In AMD architecture [38], up to two address ranges of varying sizes can be controlled using IORRs. Figure 2.1 shows an example that maps an area of system RAM between 4GB and 5GB into MMIO using one IORR.

Each IORR has a pair of registers, *IORR base register* and *IORR mask register*. The IORR mask register contains the length of the region and a valid bit indicating whether the IORR configuration pair is active. IORR base register contains the starting address of the IORR region, as well as two important flag bits, *WrMem* and *RdMem* [38]. When these two bits are set to 1, the north bridge directs read/write requests for this physical address range to system memory. When these bits are

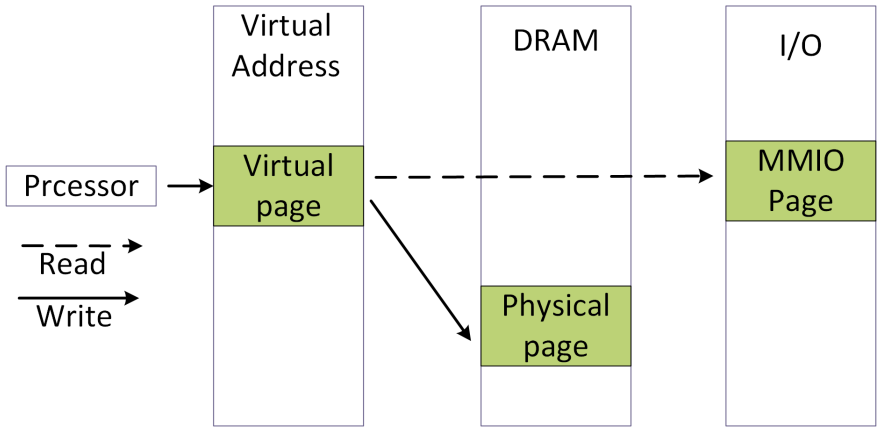

Figure 2.3: Blackbox Write

cleared to 0, all reads/write requests are directed to memory-mapped I/O.

The RdMem and WrMem bits in IORR are originally designed for shadowing ROMs of I/O devices in DRAM memory to improve system performance. The system can create a shadow region by setting $WrMem = 1$ and $RdMem = 0$ for a dedicated memory range and then copy the ROM from I/O device into DRAM memory. Once the copy operation is completed, the system changes the bit value to $WrMem = 0$ and $RdMem = 1$. Now the memory reads are directed to the faster copy in the DRAM memory instead of ROM of the device; write requests are still being directed to the ROM, but the ROM simply ignores any write request.

The I/O shadowing provided by IORRs can be misused to redirect processor requests of a valid system memory area to the I/O space. When both RdMem and WrMem bits are set to 0 in the IORR, all read and write requests to the HIveS memory will be redirected to the I/O space. With this configuration, the HIveS memory becomes inaccessible for all processor cores. Since both Windows and Linux operating systems make no assumptions on the default configurations and usages of IORRs, the modification of unused IORR registers has no impact on the OS reliability. In addition, IORR registers offers great adaptability in both the location and size of the HIveS memory.

2.3.2 Exclusive Access in the Unlocked State

The HIveS memory in the unlocked state is designed to allow an exclusive access from the processor core controlled by the attacker, while preventing acquisition by the processor cores that perform memory forensics. IORRs are registers shared by all processor cores, so any modification on one IORR register affects all the processor cores in the system. When an attackers needs to access the HIveS memory in a single core system, she can simply unlock HIveS memory by disabling the I/O shadowing, read or modify contents in the HIveS memory, and then lock it by enabling the I/O shadowing. However, it becomes a challenge to ensure an exclusive access to

Algorithm 1: TLB Camouflage

begin

allocate a new memory page;
pause all other running cores;
all cores flush TLBs;
modify the new page PTE to point to the HIveS memory;
malicious core read/write the virtual address;
malicious core TLB entry loaded;
modify the new page PTE back to regular address;
resume all other cores;

end

HIveS memory with parallel execution in a multi-core system, since the forensic examiner can be collecting memory with the other running core. We develop two new techniques, *Blackbox Write* and *TLB Camouflage*, to solve this problem.

Blackbox Write

When an attacker with an active keylogger uses HIveS memory to store the collected sensitive data, it will be writing to the HIveS memory most of the time and does not need to frequently read it back. On the other hand, forensic examiners are interested only on reading the memory contents. In order to preserve the integrity of the evidence, memory forensic tools always read the memory contents and never write to the memory.

Based on the above asymmetric operations between the attackers and the examiners, we develop *Blackbox Write* to redirect all memory read requests to the I/O space by setting $RdMem = 0$ in IORR and send all the memory write requests to the HIveS memory by setting $WrMem = 1$. With this setting, attackers can write new contents into the HIveS memory while preventing forensic examiners from reading and analyzing it. Because there is no real I/O device in the I/O hub to respond to the memory reads, a default value (e.g, $0xFF$ in AMD FX processor [45]) is returned instead. Note the examiner can also write into the HIveS memory, however actively modifying memory is a an act of compromising evidence, which is against the principle of digital forensics.

The attacker eventually needs to send the data in the HIveS memory to a remote machine. For instance, after a fix amount of user key strokes have been stealthily recorded, the keylogger can send the data to a remote server using network packets. Instead of unlocking processor's read access to the HIveS memory, the attacker can manage to read the HIveS memory by a peripheral device via DMA. To prevent random peripheral devices from reading the HIveS memory, HIveS can set the IOMMU to only allow a designated peripheral device to access the HIveS memory. Thus, a network interface adapter can read the key stoke logs from the HIveS memory via DMA and exfiltrate them.

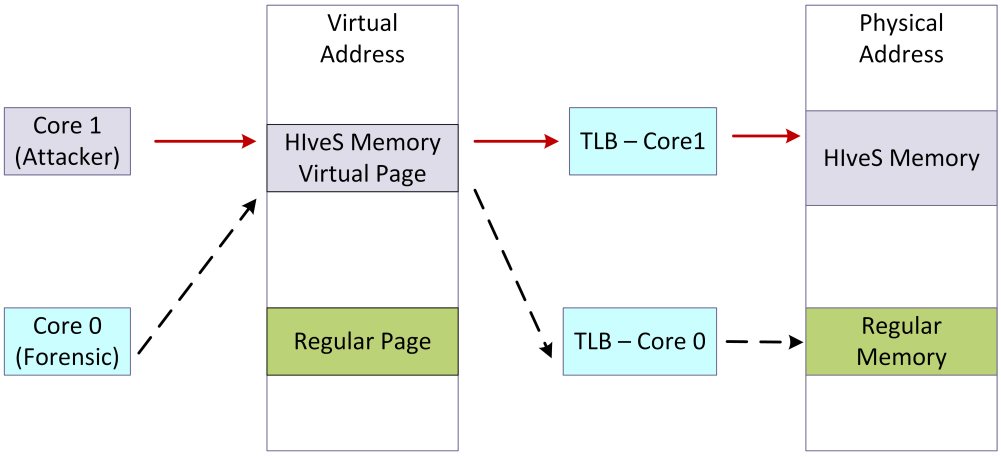

Figure 2.4: TLB Camouflage

TLB Camouflage

Blackbox write is an effective technique for malware that continuously stores sensitive data in a secret place with little need to read back, such as keyloggers. However, when the malware needs to unlock the HIveS memory for continuous read and write, it leaves a large time window for memory forensic tools to acquire the HIveS memory. We propose TLB Camouflage technique to mitigate this problem. Figure 2.4 shows the basic idea of TLB camouflage, where the unlocked HIveS memory can only be accessed by the malicious Core 1 that is controlled by the attacker, while the read and write requests from Core 2 for memory forensics are redirected to another memory space. TLB camouflage enables exclusive access to HIveS memory by creating an incoherent view of memory mapping between cores, allowing the HIveS memory contents be accessed only by the processor core that is running the malicious software.

Modern operating systems enable paging mechanism to translate virtual memory address into physical memory address before passing the memory access request to DRAM Controller (DCT) [68]. Translation-Lookaside Buffer (TLB), also known as page-translation caches, is designed to reduce the performance penalty during the time-consuming address translation process [38]. Only one memory access per virtual memory request is required when the translation for the demanding page is present in the TLB (a TLB hit). When there is no entry in the TLB for the demanding page, a TLB miss occurs. And the translation information for the page is copied from a page table entry (PTE) into the TLB (a TLB reload).

Each processor core has its own TLB [38, 35]. When the operating system changes a page mapping, the TLB won't be automatically updated to reflect the new virtual to physical address translation. TLB camouflage exploits this property to create a page translation incoherence among different processor cores.

A pseudo code of TLB Camouflage is shown in Algorithm 1. The idea is to create an incoherent cache entry in the TLB caches among the running processor cores. A new page is allocated in the

kernel for the page translation manipulation, such that the rest of the system would not be affected. Then, all the other processor cores are paused. At this point, the malicious core can flush the TLB and to make sure that there is no preexisting translation stored for the our newly allocated page already. The PTE of the allocated page is then modified to point to HIveS memory, and several LDR instructions are then used to force a translation table walk and TLB reload. And the malicious core would have a TLB entry mapping to HIveS memory. Then the PTE is modified back to the original values, and the other cores are resumed. Technically, the TLB entry for the allocated page of malicious core is incoherent, and contains a false mapping. And this is exactly what we need. In Figure 2.4, when Core 2 requests to access the virtual page of the HIveS memory, it will get the content in the regular memory. On the other hand, since the malicious Core 1 has an incoherrent TLB entry pointed to the HIveS memory address, it can access the HIveS memory if the TLB entry has not been flushed out.

TLB camouflage technique greatly increases the usability of HIveS memory, which can be used not only as a temporary storage with few interactions, but also as an interactive memory storage to support more malicious operations.

However, TLB Camouflage has some limitations. First, not all forensic tools rely on the existing kernel page tables to map virtual addresses to physical addresses [175]. In this case, TLB camouflage won't work since it depends on manipulating the kernel page tables. Second, the TLB entry should be sustained all the time; otherwise, the malicious core cannot access the HIveS memory either. Since TLB locking capability is not supported by the latest x86 architecture, malicious code has to freshen the TLB entry periodically in order to keep it active in TLB depending on the processor implementation.

2.3.3 HIveS Memory Access Property

When the HIveS memory is in the locked state by applying the I/O shadowing technique, none of processor cores can read or write the HIveS memory. Most of the time, the attacker does not need to access the HIveS memory at all, so it can lock the memory for a better protection. However, the attacker has to unlock the memory eventually to access it. When the attacker only needs to write to the HIveS memory, it can use the Blackbox Write technique. Moreover, if the attacker also needs to frequently read the memory contents, it can use the TLB Camouflage technique. Table 2.1 shows the different access privileges to the HIveS memory for both attackers and forensic examiners when applying different anti-forensic techniques.

HIveS is operating system agnostic, so the HIveS memory can be concealed on x86 platforms for both Windows and Linux. However, we need to develop a kernel module on Linux or a device driver on Windows with the root privilege to set the hardware registers. Contrary to current rootkits that modify kernel data structures or routines in the operating system, HIveS does not leave any trace in the memory or hard disk, so it cannot be detected by checking the integrity of the OS image in the memory and the hard disk.

Table 2.1: Comparison of Access to HIveS Memory

	Attacker Read	Attacker Write	Regular Read	Regular Write
I/O Shadowing	no	no	no	no
Blackbox Write	no	yes	no	yes
TLB Camouflage	yes	yes	no	no

2.4 HIveS Extension

HIveS is mainly developed to defeat software based memory acquisition methods that rely on a trusted software module in the operating system to acquire the physical memory through the processor to memory interface. Both I/O Shadowing and Blackbox Write rely on modifying the IORR registers, and TLB Camouflage creates an incoherent page translation in TLB caches of multiple processor cores. All the modifications are made on the processor, and thus only affect processing of memory request originated from the processor. On the other hand, hardware based memory acquisition solutions can detect HIveS, since a dedicated I/O device can capture physical memory image via direct memory access, which totally bypasses the processor hardware configurations made by HIveS. Moreover, the Cold Boot technique [111] exploits the physical remanence property of memory chips to directly extract sensitive data from the chips. Cold boot technique resets the system and invalidates all configurations prior to system reset. To enhance the capability of HIveS against the hardware based forensics tools, we propose to retrofit a number of existing techniques in HIveS, including IOMMU, RAM-less encryption, and Cache based I/O storage.

2.4.1 Hiding from I/O Devices

We propose to use IOMMU to evade physical memory forensics by I/O devices via DMA. Similar to the translation from virtual memory address to physical memory address performed by MMU, IOMMU is a hardware device that translates device DMA addresses into proper physical memory addresses. Each I/O device is assigned a protected domain with a set of I/O page tables that define the corresponding memory addresses. During a DMA transfer, the IOMMU intercepts the access message from the I/O bus and checks its cache (IOTLB) for the I/O to memory address translation along with the access right. IOMMU is controlled with in-memory tables and memory-mapped registers. Once a DMA request passes IOMMU, it is then processed by the north bridge. The north bridge then forwards the request either to the I/O hub or the DRAM controller base on the ranges defined by DRAM Base/Limit and MMIO Base/Limit registers. Therefore, HIveS can set the IOMMU to only allow a peripheral device to perform DMA into assigned regions, thus preventing a full system memory acquisition with DMA. When the IOMMU is not available on some old systems, the DMA can also be redirected by manipulating the north bridge using MMIO Base/Limit registers. The main idea is to modify the MMIO Base/Limit registers to bounce DMA

reads back to the I/O hub. The details can be found in [161].

2.4.2 Hiding from Cold Boot

There are two solutions to evade Cold Boot based memory acquisition mechanisms: *RAM-less encryption* and *Cache based I/O storage*.

RAM-less encryption. The basic idea is that attacker encrypts all the memory contents in the HIveS memory with a secret key stored in CPU registers [146, 170]. Since operating systems do not use all the MTRR and IORR register pairs all the time, HIveS can encrypt the HIveS memory using AES and store the encryption key in unused MTRR or IORR registers. Thus, even if the physical memory is completely acquired through Cold Boot, the contents of HIveS are still being protected, because the encryption key in the CPU registers is lost forever due to the system reset.

Cache based I/O storage. The idea is to save a small HIveS memory only in the CPU cache [139, 152, 108] and then mask it with I/O Shadowing technique. When the memory address is set to cacheable in the page table entry and both RdMem and WrMem bits in the IORR base register are set to 1, any write to that location will trigger a cache line fill if the memory contents are not yet loaded in the cache. When the HIveS system is unlocked, the attacker can simply write data into memory as usual. When the HIveS system is locked, the HIveS memory is cached and masked by I/O shadowing. Therefore, neither I/O devices nor the processor can read out the HIveS memory in the cache via DMA. However, it remains a challenge to maintain the contents in the cache considering the limited cache control provided by the x86 architecture [38, 35].

2.5 Implementation and Evaluation

We build a prototype of HIveS on an x86 desktop with AMD FX processor. The motherboard is ASUS M4 A96 R2.0, running a AMD FX-8320 8-core processor with single bank DDR3 4GB memory. The 4GB memory is relatively small but it shortens the time for memory acquisition and it is large enough to demonstrate all the functionality of HIveS.

To illustrate the effectiveness of the HIveS memory, we implement a keylogger rootkit called *HIL* that uses HIveS memory to store the keystrokes so that the stolen information cannot be detected by memory forensics. We implement HIL prototypes on both Windows and Linux. On Ubuntu 13.04, we implement a Linux kernel module to support all the techniques in HIveS. On 64-bit Windows 7, we implement a kernel mode device driver as a keylogger and use WinDbg debugger to configure the IORR pair.

We implement I/O shadowing, Blackbox Write, and TLB Camouflage techniques and evaluate their effectiveness using a number of most updated software based memory forensic tools. We also implement RAM-less encryption and cache based I/O Storage techniques to demonstrate the capability of HIveS to evade Cold Boot based physical memory forensics.

Tool	Tool Version	OS	Examine IORR	With HIveS	Without HIveS
UnitTest	1.0	Linux	No	No Detection	Identified
LiME	1.1	Linux	No	No Detection	Identified
MemDump	1.01	Linux	No	No Detection	Identified
DD	8.13	Linux	No	No Detection	Identified
WinPmem	2.3.1	Windows	No	No Detection	Identified
Mem Marshall	1.0	Windows	No	No Detection	Identified
Memoryze	3.0	Windows	No	No Detection	Identified
Dumpit	1.3.2	Windows	No	No Detection	Identified

Table 2.2: Verification against Memory Forensic Tools

2.5.1 I/O Shadowing

Since modification of MSR require privilege mode, we implemented most of the functionalities in a kernel module. User space programs can communicate with the kernel module through *procfs* export. For I/O shadowing, the kernel module is responsible for manipulating the IORR register to set the base and the size of the HIveS memory, as well as the WrMem and RdMem flag bits. With the physical address and HIveS running mode passed in through procfs, the module first masks off the lower 12 bit of the physical address, and inserts it into bits 12 to 47 in I/O Range Base register, *MSRC001_0016*, since the physical addressing in AMD x64 is 47 bit. The bits 3 and 4 of the register are RdMem bit and WrMem bit respectively. For I/O Shadowing, we clear both bit 3 and bit 4 to redirect both read and write requests into the I/O space.

The IORR base register should always be written first, since the IORR mask register, *MSRC001_0017*, contains a *valid* bit, which will immediately enable the IORR pair once this bit is set. Therefore, we cannot set the two IORR registers in reverse order; otherwise, the system will fail and hang itself. In the AMD FX system [38], the valid bit is bit 11 of the IORR mask register.

Although the detailed HIveS implementation is different on Linux and Windows, the workflow remains the same. We first load HIveS as a kernel module in the system. An 1MB area at physical address offset of *0x10c800000* is allocated to be the HIveS memory. With RdMem and WrMem both set, we fill the memory with repeating pattern of *0x12345678*. Once the pattern is written, we flush the cache to make sure that these patterns are written into the memory. Then we enable I/O shadowing to lock the HIveS memory by clearing both the WrMem and RdMem bits. At this point, all the contents in the HIveS memory should be protected against memory forensic tools.

We verify that none of the software based memory forensic tools that we tested is able to capture the HIveS memory protected by the I/O shadowing technique. Table 2.2 summarizes the tools that we use in our experiments. Tools such as the Linux base memory acquisition tool LiME [180] obtain the physical memory layout of the system by parsing kernel data structure. For each valid memory region, it calls $kmap$ function to map virtual pages to physical pages to read the content.

Table 2.2 shows that none of these memory forensic tools can detect the HIveS memory through searching the special repeating pattern *0x12345678* when the I/O shadowing is enabled. However, when the memory dumps are taken again after the I/O shadowing is disabled, we can identify the repeating pattern in the memory dumps. None of the tools can detect HIveS, since all the processor memory accesses can be manipulated by the IORR registers and all the software based tools utilize processor to access memory.

2.5.2 Blackbox Write and TLB Camouflage

Blackbox Write only provides write access to the HIveS memory and prevents any read access. We implement it by clearing the RdMem bit and setting the WrMem bit. To disable Blackbox Write, we simply clear the valid bit of the IORR pair. To verify its effectiveness, we set up the keylogger to work in the Blackbox Write mode. Instead of filling the HIveS memory with repeating pattern *0x12345678*. We run the keylogger, and manually type in "this is a HIveS blackbox write test!". When Blackbox Write is enabled, we dump the memory using the memory acquisition tools, including LiME, MemDump, and WinPmem, to capture the entire physical memory images. And we verify that the sentence we typed was not found in the acquired memory image. Immediately after the first round of memory dump, we disable Blackbox Write to allow both read and write access to the HIveS memory and perform memory dumping again. This time, we were able to find the logs of what we just typed.

TLB camouflage protects the HIveS memory by only allowing read and write access to a single processor core. After pausing all other cores, we flush the TLBs of all cores. Next, we disable all interrupts on the malicious core and then read the contents of the HIveS memory into a temporary memory space. The kernel module then goes in a busy loop accessing the memory location continuously to sustain the TLB entry in the malicious core's TLB. We confirm that only a single processor core can access the HIveS memory by dumping the memory images using different processor cores and searching the coded repeating pattern.

2.5.3 RAM-less Encryption

For RAM-less encryption, we use a secret key to XOR the plain-text instead of using the AES function, since the feasibility of RAM-less encryption has already been verified [146, 170] and our focus is on testing the stability of the MSRs for storing the secret key. In particular, we use the unused MTRR registers and IORR registers, which can be identified by checking the valid bit. On our AMD platform, there are eight MTRR pairs per core plus two shared core IORR registers. When the valid bit is cleared, the register is not used by the system. The bits provided by these registers are large enough to store a short encryption key.

2.5.4 Cache based I/O Storage

We perform a simple experiment to verify that the cache based I/O storage is able to keep the sensitive data in the cache only. Similarly, a repeating pattern *0x12345678* is written into the HIveS memory. Now the pattern should be stored in the cache. Next, we execute an *INVD* instruction, which invalidates all cache content without writing them back to the physical memory. If the pattern is indeed in the cache, after the execution of *INVD* instruction, such written pattern should no longer be observable. In our experiment, since the memory read back after *INVD* is not *0x12345678*, and therefore the modifications to the memory we wrote was truly stored in the cache. However, when the processor is busy, such contents stored in the cache is flushed out to the physical DRAM in a very short time.

2.6 HIveS Limitations and Countermeasures

2.6.1 HIveS Limitations

Though the prototype shows promising potential on using HIveS to conceal malicious code and sensitive data in HIveS memory, the system has some limitations.

First, since the basic idea behind HIveS is the manipulation of physical address layout, system architecture with a fixed or reliable way to retrieve the physical address layout is not vulnerable to this attack. Furthermore, our implementation of HIveS relies on manipulating hardware registers in the AMD processor [38], therefore porting of the malware to other platforms requires careful design changes and examinations. As with most other advanced anti-forensic rootkits, HIveS requires kernel privilege to manipulate the system registers, and thus it is not available to user space malwares.

Second, HIveS achieves stealthiness by redirecting memory access on the hardware level. This inherently implies all software based accesses to the memory are redirected. Therefore HIveS cannot be used to store the current executing code. As a result, malware code that utilizes this storage, as presented in our prototype, is left in the memory and could be captured and analyzed by forensic examiners.

Lastly, HIveS focuses on defeating the software based memory acquisition approaches, so it has to be augmented with other anti-forensic mechanisms to defeat the hardware based memory acquisition approaches. Those mechanisms increase the complexity of HIveS and possibly make the targeted system unstable.

2.6.2 Countermeasures

HIveS is a system to subvert the organization of physical address layout. In order to defeat HIveS, it is important to get a reliable representation of the true address layout. Unfortunately, there is currently no architecturally supported method to verify the truthfulness of the layout. For the rest of the discussion, we focus on how to defeat our HIveS implementation on AMD platforms.

First, we know that the manipulation of IORR is essential in HIveS, and since IORR registers are only available on AMD processors, HIveS as a system does not work well in Intel family processor from HIveS, though several techniques we presented in HIveS may still apply.

Second, the use of IORR can be a good hint of the presence of HIveS memory in the system. It can be identified by simply inspecting the valid bit in the IORR mask register. A forensic examiner can also detect the existence of HIveS by measuring the timing for memory operations. When I/O shadowing is enabled, all memory access goes through the I/O bus and takes much longer to complete.

Note that legitimate I/O devices may also use the IORR to map physical memory address to the I/O space. For instance, AGP video driver in Linux kernel uses the IORR register in some cases. Since AMD provides two pairs of IORR registers, a forensic examiner can also examine the difference of the two pairs. Yet, such analysis could be quite system dependent. It is difficult to determine if the use of IORR registers is benign or malicious. The forensic examiner may assume the use of IORR is malicious and directly modify the value of this IORR register to reveal the memory contents. However, if the use of IORR is benign and system originated, such direct manipulation may crash the system. A more conservative approach might be to read out the contents from the address range without modifying any of the registers. Even though this might cause system instability as well, the probability of a system catastrophic failure caused by memory read is much smaller. If all the bytes read back are all identical values $0xFF$ or $0x00$, then most likely there is no real I/O device behind these I/O addresses.

Finally, HIveS can be detected by Cold Boot if we don't apply the HIveS extensions such as RAM-less encryption. Forensic examiners can first dump the registers, including all the MSRs and debugging registers from all processor cores. All system cache can then be flushed back into memory. Then the system is reset to extract memory content exploiting the memory remanence characteristics. This however changes many system configurations in the system as well as some memory contents, which violates the forensic principle of not altering the crime scene.

2.7 Extension and Future Work

While some of the limitations discussed above are unavoidable, such as dependence on architecture and operating system, some others can be overcome. We discuss possible extensions of HIveS in this section.

2.7.1 Eliminating Memory Traces

One of the limitations discussed above is that the storage can only be used to store data collected by the malware instead of protecting the entire malware. The forensic examiner might be able to analyze the malware memory to discover the manipulation of address layout. One key insight is that, the physical address layout manipulation performed by HIveS is very infrequent if not one time. Furthermore, the amount of code required to alter this layout is very small, most likely a single instruction or two. Taking our prototype as an example, the kernel module initialization routine can use one instruction to change the IORR register then immediately erasing the previous instruction by zeroing it. This leaves a very small time window, three instruction execution time, for the forensic examiner to capture the image and discover the use of IORR in the malware. The practical chance of catching such moment is close to zero.

2.7.2 Extending HIveS to Intel Platforms

The IORR registers HIveS exploited to alter the physical address layout is AMD specific unfortunately. To the best of our knowledge, there is no such MSR in the Intel platform [35]. This does not imply HIveS is impossible on Intel. Malware authors will need to find another way to alter the physical address layout to launch the attack. For example, Intel Memory Controller Hub (MCH) chipsets also provide capability to recover addressable memory space lost to MMIO space [25]. One can modify the REMAPBASE and REMAPLIMIT register in the chipset to manipulate the physical address layout (also known as system address space in Intel manuals).

2.7.3 HIveS for Defense

Techniques in computer security are like weapons, it can be used either to defend the righteousness or cause damage to the society. For instance, virtual machine based rootkit (VMBR) introduced by Rutkowska et al. [162] has been used to capture host image in forensic memory analysis [141, 204]. Similarly, though we present HIveS as a powerful anti-forensic tool, it can certainly be developed and used as a defense tool to protect sensitive data against malicious memory scanning. For example, application passwords can be stored in HIveS memory without having to worry about malware reading the passwords from the physical memory.

2.8 Related Works

There is an ongoing arms race between the attackers and the forensic examiners in computer forensics [172, 113, 148, 190]. Memory forensic analysis is becoming an indispensable tool for forensic examiners nowadays, and they have two ways to acquire computer memory: software

based methods that use a trusted software module to access memory through the CPU processor [70, 133, 194, 158, 175, 89, 166, 180, 76] and hardware based methods that rely on dedicated I/O devices to access physical memory image via Direct Memory Access [69, 175, 154, 64].

Software based memory acquisition techniques rely on the CPU processor to acquire physical memory through the operating system. Unfortunately, after recognizing this dependency, attackers have developed anti-forensic techniques to compromise the memory acquisition process, such as directly modifying the acquisition module or the OS kernel data structure [66, 113, 172, 91], using rootkits to hook operating system APIs [168], or installing a thin hypervisor on the fly [162].

To defeat those anti-forensic techniques, Stüttgen et al. [175] propose an anti-forensic resilient method to acquire physical memory by eliminating its dependence on the operating system routines and data structures. Schatze [166] proposes to bootstrap a trusted new execution environment from the normal one to make sure that the operating system is free of malware. System management mode (SMM) can also be used to create a trusted isolated execution environment [158, 194]. Some researchers propose to go deeper than the operating system level and use hardware virtualization to avoid the memory acquisition software being subverted by rootkits [141, 204].

Stüttgen et al. suggest that the memory acquisition process can be trusted if the acquisition module has not been tampered with and all the operations are performed without relying on the operating system or any other untrusted software [175]. However, in this work, we show that this assumption is not true. The main reason is that the physical memory layout seen by the processor can be manipulated through the hardware configurations on the chipset. Attackers can misuse hardware configurations to modify this layout and conceal the presence of malware.

A number of hardware based memory acquisition methods have been developed recently [154, 193, 63, 69], using a trusted peripheral device to capture the physical memory image via DMA. Since it does not rely on the CPU processor to get the physical memory, the hardware based approaches can successfully prevent those anti-forensic techniques that are originally designed to defeat the software based approaches. However, Rutkowska [161] shows that it is possible to present a different view of the physical memory to the peripherals by reprogramming the north bridge. Therefore, in-memory data acquired by DMAs could be compromised as well [158, 175].

A special type of memory acquisition technique relies on the unique remanence property of physical DRAM [111, 70]. Despite the popular belief that volatile contents in DRAM are gone once the computer resets or powers off, Halderman et al. [111] demonstrate a Cold Boot attack that can reliably recover the contents in the memory modules even after the power has been cut off for a short period of time. Though the original Cold Boot is demonstrated as an attack to steal cryptographic keys and other sensitive data from the RAM, it is also an effective method that can be used for reliably acquiring physical memory.

2.9 Summary

In this work, we propose a different approach to anti-memory forensic. Instead of looking at ways to conceal presence by operating system object manipulation, we can defeat current memory acquisition methods by manipulating the physical address layout, a design architectural feature on modern x86 platforms.

HIveS is an anti-forensic mechanism to conceal in-memory data shadowed behind the I/O address space. Besides I/O Shadowing technique to prevent forensic memory acquisition tools from reading the HIveS memory contents via processor, we also use Blackbox Write and TLB Camouflage to enable the attacker exclusive write access and provide a single malicious core exclusive read and write access, respectively. Furthermore, we propose several add-ons to the basic framework to further hide from physical memory forensics.

A prototype of HIveS is built on an AMD platform to show that none of the popular memory acquisition tools we tested can capture the memory data protected by HIveS. Several countermeasures are discussed in the end. In the future, we intend to further investigate possible mechanisms to retrieve trustworthy physical address layout.

Chapter 3

Attack - CacheKit: Exploiting Cache Incoherence to Evade System Introspection from Trusted Domain on ARM

3.1 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 [92, 9, 81].

ARM family processors are currently the most widely used processors, powering over 60% of all embedded devices [6, 40], including 4.5 billion mobile phones [41]. *TrustZone* is a hardware security extension offered by ARM to provide an isolated trusted execution environment [46]. 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 [140, 132, 212, 201, 165]. 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 [52, 176].

In this work, 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 [27]. 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 [124, 3, 167, 23, 58, 24, 162, 86].

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 [94] 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 chapter is organized as follows. Section 3.2 describes some background on ARM TrustZone and ARM cache architecture. We present the CacheKit architecture in Section 3.3. A prototype implementation is detailed in Section 3.4. We evaluate CacheKit in Section 3.5 and present potential countermeasures against CacheKit in Section 3.6. We discuss the impacts of the new rootkit in Section 3.7. Section 3.8 discusses related works. Finally, we conclude the chapter in Section 3.9.

3.2 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.

3.2.1 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 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.

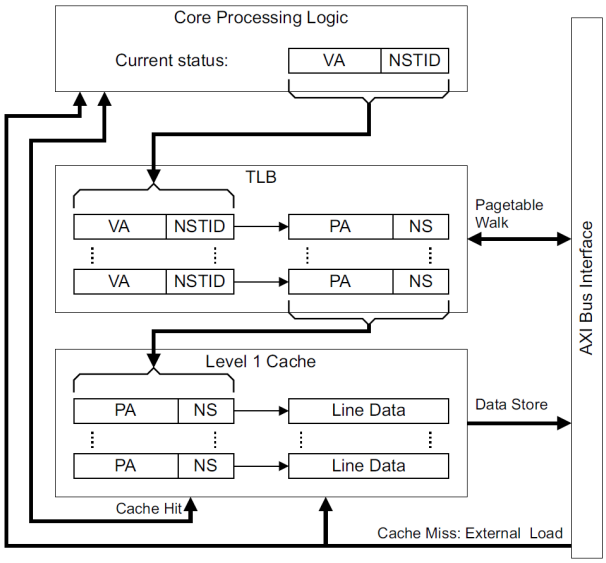

Figure 3.1: Caching in Trustzone [27]

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.

3.2.2 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 [27]. 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 [27]. 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 3.1 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

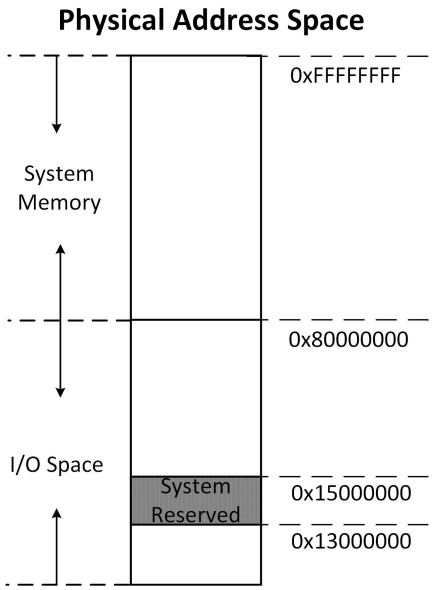

Figure 3.2: Physical Addressing on 32bit ARM System

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.

3.2.3 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. 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 3.2. 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.

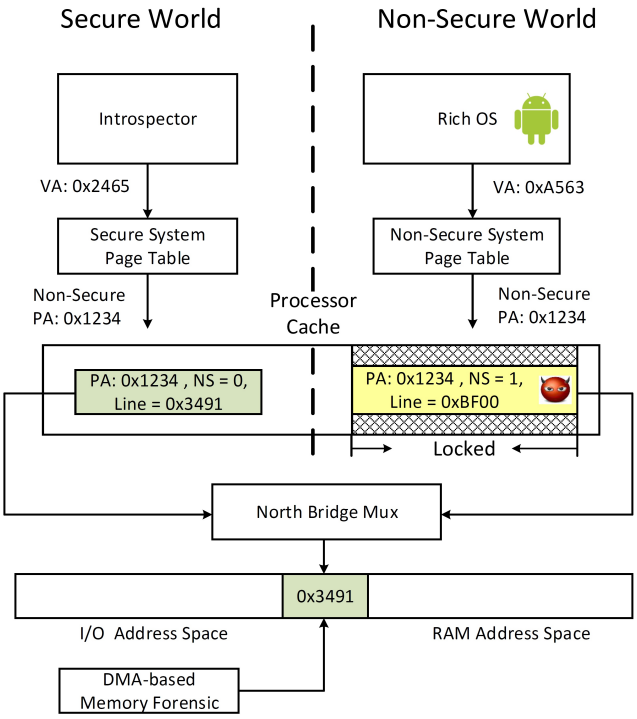

Figure 3.3: CacheKit Architecture

3.3 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 3.3. 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.

3.3.1 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 [27]. 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 3.4, 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 $= 1$.

3.3.2 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.

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 [138, 139] 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*. There-

fore, 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).

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 [29], CortexA8 [31], CortexA9 [78], and NViDIA Tegra [36]. 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 [125]. 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 3.4.

Cache Concealing

CacheKit must evade introspections from both the secure world and the normal world. Many modern rootkit analysis tools [16, 22] 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 [11] 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

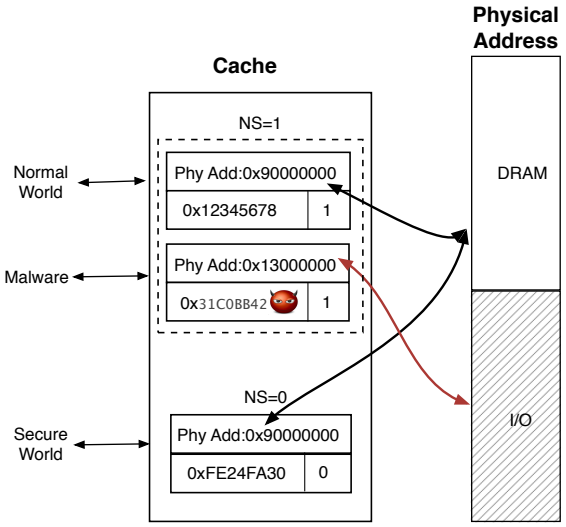

Figure 3.4: CacheKit Overall Design

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.

3.4 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.

The overall design of CacheKit is shown in Figure 3.4. 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.

3.4.1 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.

Table 3.1: Trustzone Cache Behavior

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

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 3.1 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 3.1 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 3.1) to those with the NS bit set (i.e., the last two rows in Table 3.1). 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 3.1 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.

3.4.2 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.

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.

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 [125].

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 [11] 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.

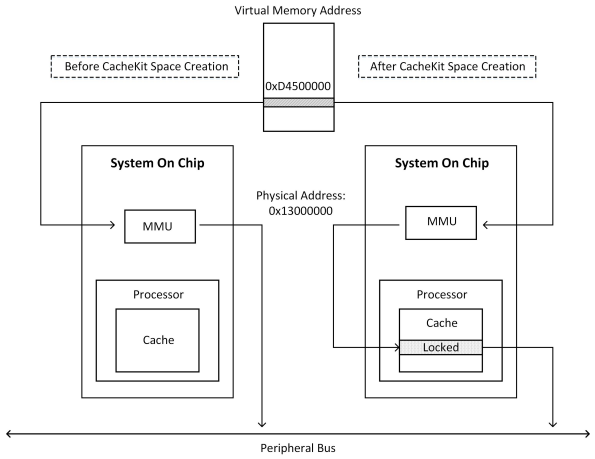

Figure 3.5: CacheKit on I/O Address Space

To resolve the two problems with direct cache locking, we propose to carve a piece of usable memory space out of 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 3.5 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.

3.4.3 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-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 Sup-

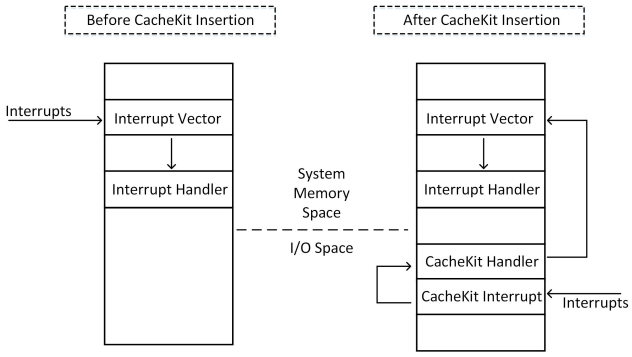

Figure 3.6: IVT CacheKit

port 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 [82] 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 3.6, 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.

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 3.2: CacheKit Benchmarks on Code Sizes

3.5 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.

3.5.1 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.

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 [30, 38, 35]. 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.

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 [110] 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.

Cache Elusiveness

We evaluate the cache elusiveness against TrustDump [176], 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 [11] to dump the physical memory of the experimental platform onto the SD Card. We then perform a binary pattern search on the 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.

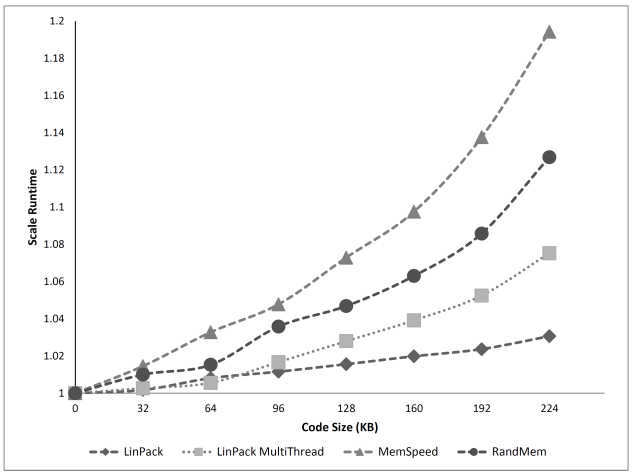

Figure 3.7: CacheKit Performance Impacts

3.5.2 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 [84], MemSpeed and RandMem [137]. 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 3.2 shows the performance impacts with various code sizes.

The columns in Table 3.2 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 3.7 and Table 3.2 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

Table 3.3: Rootkit Techniques and Detection Comparison

Detection Methods	User	Kernel	VMBR[162]	SMMR[86]	Cloaker[82]	CacheKit
App level detection	✓	×	×	✓	×	×
OS level detection	✓	×	×	×	×	×
VMM level detection	✓	✓	✓	×	×	×
Coprocessor Based	✓	✓	✓	×	×	×
TEE Based	✓	✓	✓	×	×	×
Physical Memory Check	✓	✓	✓	✓	✓	×

compare baseline with the maximal size rootkit and observe a 12% overall performance overhead.

3.6 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. 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.

3.6.1 Evading Detection

A comparison between CacheKit and other types of rootkits against existing detection methods is provided in Table 3.3. Researchers propose to use the trusted execution environment (TEE) provided by hardware to perform the detection of rootkits in the memory [52, 193, 144, 163, 176]. 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 [198, 213, 195, 59] 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 [96, 90], it can simply stop all malicious activities. CacheKit can be augmented with this evasion logic to remain undetected.

Coprocessor-based detectors [154, 211] 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 [111, 147, 70] 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.

3.6.2 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* [127].

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.

3.7 Discussion

3.7.1 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 [52, 100]. 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 [52]. 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.

3.7.2 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 [24, 58, 209, 167, 82, 162] 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 [31, 78], 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 [209, 82, 162].

Furthermore, with the current always-connected network architectures, attackers have other means to obtain persistence such as through watering hole attacks [106]. 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.

3.7.3 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 [47].

Another closely related problem is the cache space optimization while applying CacheKit to existing rootkits. For example, the rootkit adore [24] 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.

3.7.4 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 [61].

3.8 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, 124] can effectively detect them.

Later, malware authors developed in-memory rootkits that reside only in the operating system kernel memory [24, 58, 23, 167] to defeat the storage-based detection. To insert itself into the kernel control flow, in-memory rootkits either hook into legitimate kernel functions [18, 83, 129] or modify kernel data structures [23, 24, 58]. 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 [198, 213, 195, 59]. To enable offline rootkit analysis, researchers also propose to acquire the system memory using a dedicated secure coprocessor [154, 211] or physical hardware [111, 147, 70].

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 [162, 86, 114]. Virtual machine based rootkits (VMBR) [162] insert a customized malicious hypervisor beneath the currently running operating system. Firmware based rootkits infect the firmware on I/O devices [114] or the system BIOS [86]. However, rootkits are not alone in seeking higher privilege. New hardware and software rootkit detectors with higher privilege are also proposed [51, 98, 153, 52, 53, 193, 144, 163, 156, 122, 149, 154, 211].

Hypervisor is used in several research to perform introspection into the guest operating system [51, 98, 153]. However, since the virtualized environments can be reliably identified [96], malware can simply exit without execution. Moreover, new vulnerabilities are frequently found in the hypervisors [90]. Hardware features, such as security extensions in various processors (e.g., AMD SVM [38], Intel TXT [185], and ARM TrustZone [46]) 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 [52, 193, 144, 163, 176, 187]. On the other hand, hardware features can also be exploited by rootkits to evade detection in the OS kernel [82, 172, 209]. For instance, Shadow Walker [172] exploits the *I-TLB* and *D-TLB* coherency problem in the Intel architecture to hide the rootkits. Cloaker [82] can hide the location of the replaced interrupt descriptor vector by locking the page translation in the translation look aside buffer.

In this work, 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.

3.9 Summary

In this work, 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.

Chapter 4

Defense - TUDEC: Enabling Trusted Data-Intensive Execution in Untrusted Cloud Computing

4.1 Introduction

Cloud computing is driving a new paradigm shift in modern world computing. With virtualization as the key enabling technology, service providers are now able to dynamically provision the computing resources to meet the ever changing demands from users. In addition, such economy of scale has provided opportunities for better service at a lower price. However, even with such economic incentives, cloud computing has yet to become the dominating computing paradigm. The most often cited and widely acknowledged concern is the security and privacy of user data in the cloud, due to the fact that the cloud service providers are considered third-party with respect to the data owners.

Securing user sensitive data in the cloud against "honest-but-curious" cloud servers has been a very active research topic in the past several years [192, 75, 205, 179, 191, 203]. An encryption-before-outsourcing approach has been proposed in order to provide user data privacy while data are stored in the cloud. A significant amount of research effort has been devoted to protect the security and privacy of cloud data storage, ranging from public auditing of data storage in the cloud [192], encryption-based access control and data sharing [205], to search over encrypted data [179, 131]. On the other hand, most of the applications in the cloud involve data-intensive computations, which require highly efficient secure computation mechanisms. Despite the recent breakthrough in fully homomorphic encryption system which enables arbitrary computation over encrypted data, the complexity of this system is forbiddingly high to be adopted in practice. Other secure computation outsourcing efforts only lead to solutions applicable to some specific types of computations [73, 199, 50].

In this work, contrary to the existing cryptography-based secure computation outsourcing solutions, we take a completely different approach to securing data computation in the untrusted third party cloud server. The idea is to provide a trustworthy execution environment so that user data can be decrypted and processed in plain text inside this container while maintaining confidentiality of the information against curious cloud host. With this protection, user data are encrypted at rest (i.e. when stored) but decrypted during runtime, significantly enhancing the efficiency. Our system is built with a small set of trusted computing base, comprised of only firmware and hardware, aimed to secure the execution environment against software-based attacks from either cloud system administrators or other VM users. This challenge has two major aspects: the environment has to be both secure and data efficient.

In order to secure the environment, both the virtualization software and hardware underneath have to be trustworthy. Virtualization software is the foundational technology that enables sharing of physical resources while enforcing isolation between virtual machines (VM) of different users in the cloud. It has been generally accepted that there is risk of attacks on virtualization software stack, from malicious guest VMs [37, 12, 157, 128]. Adversaries can also compromise the cloud management softwares [37, 12], or even use social engineering to obtain the management credentials. Given the advanced cyber attacks happening daily, it is necessary to guarantee the security and privacy of user data even if the cloud infrastructure becomes compromised by adversaries. Recognizing this problem, trusted execution environment has recently been an active area of research [54, 144, 178]. Solutions along this line generally rely on a trusted platform module (TPM) to provide a root of trust of the system in the presence of an adversary. To be data efficient, the environment must be able to process a large amount of data with little performance overhead in the virtualized environment. Data processing in such environments involves heavy use of virtualized I/O devices, which is the current bottleneck of system virtualization [135, 104].

In this work, we propose TUDEC, a **T**r**U**sted **D**ata-intensive **E**xe**C**ution environment optimized for data intensive applications in the cloud. TUDEC aims to provide a general solution to the security and privacy of user data while data are being processed in the cloud. Our solution is not specific to any particular type of computation. In addition, It is efficient for large data processing, which depends heavily on the I/O subsystem. In particular, TUDEC has the following two unique attributes:

- TUDEC is designed to provide an isolated, trustworthy environment. The system utilizes state-of-the-art hardware technologies such as hardware virtualization, system management mode (SMM), input output memory mapped unit (IOMMU) to dedicate resources to individual user execution environment while effectively isolating them. With such hardware enforced isolation, the system is able to offer its user the protection of her private data against software based attacks from the hypervisor as well as neighboring malicious guest VMs.
- TUDEC is designed to provide an optimized environment for data-intensive applications in the cloud. With advanced hardware-assisted virtualization, the processor computing performance in TUDEC is close to bare metal. Furthermore, by using direct device assignment

with exitless interrupt, TUDEC enables the most demanding I/O-intensive applications in a fully loaded cloud service. Compared to existing works on a secure computing environment [54, 123, 144], TUDEC offers a unique advantage in *I/O intensive data applications*.

4.2 TUDEC Overview

4.2.1 Trusted Isolated Execution Environment

One of the main goals of TUDEC is to provide an execution environment that is isolated and trustworthy. It builds on the intuition that remote cloud users should be able to bootstrap the trust in an isolated domain via remote attestation. Furthermore, this execution environment should be isolated from the rest of the system such that the trust established during initialization can be maintained. We decompose the problem into isolating the processor, memory, and I/O devices. Since TUDEC is designed for data-intensive executions, which tend to use the processor aggressively, there really is little incentive to multiplex resources for this type of virtual machines. Each virtual machine swap is bound to be very costly because of processor state manipulation and cache pollution. Therefore, each execution environment in TUDEC has its own dedicated physical resources. However, physical resource dedication is often not enough to guarantee the isolation of different executions, since x86 architecture is designed to have multiple processor cores sharing chipsets and many other components. These processor cores communicate frequently with each other and with the bus and I/O devices that we will have to isolate. In TUDEC, hardware virtualization technologies such as Secure Virtual Machine (SVM) [38] are used to facilitate this isolation. Two-level paging support, such as Nested Paging Table (NPT) [26], is used to guard against a malicious guest corrupting memory of its neighboring guest via page table manipulation, and Input/Output Memory Mapped Unit (IOMMU) and System Management Mode (SMM) [38] are used to protect against an I/O device from using Direct Memory Access (DMA) to read or write memory of the isolated environment. The details of this design will be presented in Section 4.4.

4.2.2 High Performance I/O

TUDEC aims to provide a trusted execution environment for data-intensive applications. Therefore, besides securing the environment, it is also important for the system to be I/O-efficient, since the performance of the execution that it is designed to support is closely related to the I/O operations. With the introduction of hardware-assisted processor and paging virtualization, I/O is now the bottleneck for virtualized platforms. To improve the virtualized I/O subsystem, many virtualization platforms have implemented direct device assignment, with which guests are given direct access to the I/O devices. However, it is not enough to gain performance close to bare-metal configuration by simply using direct device assignment [135, 104]. With network traffic as small as

50MBs, researchers were able to observe performance degradation [104]. This is due to the interrupt delivery processing in the hypervisor to remap interrupts. Under the current implementation of PCI passthrough of XEN [60, 155], though the guest operating system has direct access to the PCI device and the I/O device can also perform DMA on the guest's physical memory, the hypervisor is still acting as a proxy for interrupt signaling. In the current device model, when the I/O device issues an interrupt, it forces the execution of the processor core out of guest mode and traps back to hypervisor running in the host mode. The interrupts are then handled with the hypervisor IDT. After acknowledging the interrupt, the hypervisor then injects a virtual interrupt into the guest. The guest, completely unaware of this real-to-virtual interrupt mapping, treats the virtual interrupt as a normal one using the guest IDT. Upon completion of interrupt processing, the guest interrupt handler will write to the End of Interrupt (EOI) register to signal the completion of the interrupt, which will cause another VM exit. Upon trapping the VM exit, the hypervisor will emulate the completion of the virtual interrupt and resume guest processing. Therefore, accompanying every interrupt signal are two VM exits. These exits are known to be a major source of performance overhead in virtualization. They are expensive due to the processing of state switches and the cache pollution. In order to reduce this overhead, the interrupts are directly delivered into the virtual machine, and the EOI register is also directly exposed to the guest. However, this implies that the potentially untrustworthy guest is granted complete control of the Local Application Programmable Interrupt Controller (LAPIC). As illustrated in detail in Section 4.4, guests in TUDEC are granted dedicated processor cores, and we preprocess the interrupt delivery in TUDEC.

4.2.3 Securing Direct I/O Device Access

The current hardware-assisted virtualization focuses mostly on the processor virtualization, such as AMD SVM [38] and Intel VMX [35]. Memory virtualization technologies such as second level paging table like AMD NPT [26] and Intel EPT [184] were added to further improve the efficiency. Though significant efforts were made to push the I/O virtualization forward [104, 184, 28, 35, 38, 13], the sharing of I/O devices among multiple virtual machines is still mainly managed and realized by the hypervisor emulating part or all of the interactions, which makes direct device access a necessity for data-intensive applications. However, once an adversary gains access to a virtual machine with direct access to a PCI device such as a network card, she will be able to program the I/O device to perform DMA on arbitrary physical memory and thus capable of corrupting any system data structure. IOMMU is a hardware device that translates device DMA addresses into proper physical memory addresses. Though originally designed to handle the differences in addressing capability between the processor and I/O devices, IOMMU has become one of the invaluable components in virtualization. Modern IOMMUs are not only used for translation but also isolation between different DMA-capable devices and the host. Current systems rely solely on the IOMMU to carry out protection against malicious guest DMA attacks [104, 200, 174]. However an adversary with access to legacy host can alter the configuration of the IOMMU and defeat the domain protection. We propose to lock down the IOMMU configuration base address during system initialization, and map control registers into SMRAM to prevent legacy host from accessing

it.

4.3 System Model

4.3.1 System Overview

For simplicity, we assume there are three types of servers within the cloud: load balancing server, image server, and hosting server. The load balancing sever is responsible for resource allocation, the image server is a network storage for all the VM images, and the hosting server is in charge of housing the virtual machines for end users on top of the hypervisor. Each VM image is composed of two separate partial images: the system image which contains the basic operating system and the user image which contains user applications. A system image is shared among users with the same virtual platform configuration, e.g., all the users who want to run Linux. On the other hand, a user image contains user specific contents, which are considered sensitive, so it is encrypted with a secret key maintained by the user.

4.3.2 Threat Model and Assumption

We consider two types of adversaries with different control. An adversary of the first type resides outside the cloud infrastructure, namely the outsider attacker. More specifically, she has access to a virtual machine located on the same physical platform as the user. The goal of an outsider attacker is to steal information from the other guest on the same physical platform. Since it is relatively easy to gain access to a physically collocated VM [159], this attacker is not concerned about being detected and might launch a deny of service (DoS) attack on neighboring VMs. An adversary of the second type has access to the cloud hosting infrastructure, namely the insider attacker. She can be an employee working for the cloud service provider or someone who gained access through social engineering. The goal of an insider attacker is to steal documents and applications from the user VM. This attacker would not expose herself with DoS attacks.

We assume physical security will be in place such that the hardware will be protected in a controlled environment. Therefore, it would not be possible to perform any hardware level attacks, such as physically probing the bus and memory. In addition, the firmware on the I/O devices cannot be reprogrammed by the malicious system administrator to perform hardware level attacks [164]. Finally, we assume that the platform, on which TUDEC runs, is equipped with TCG's trusted boot [19] hardware.

4.4 TUDEC Design

As illustrated in Section 4.1, TUDEC is designed to provide users with a trustworthy data execution environment to enable the near bare-metal performance. User data are encrypted at rest and decrypted to operate on in the trusted isolated environment. In the remaining subsections, we will elaborate the key designs: isolation, high performance I/O, and data at rest.

4.4.1 Trusted Isolated Execution Environment

One of the main goals of TUDEC is to provide an isolated execution environment. Remote cloud users should be able to bootstrap trust via remote attestation. Additionally, this environment should be isolated from the rest of the system in order to maintain the trust state established during the environment initialization.

Trusted Startup

In order for the system to attest its integrity to a remote user, there needs to be a trust anchor upon which to extend. Trusted startup [19] is the process to establish this trust anchor in the system. There are a number of configuration registers (PCR) in each trusted platform module (TPM). In TUDEC, the trusted startup procedure will extend each trusted computing base (TCB) module loaded to the PCRs. These PCRs are later used to attest to a remote guest user the integrity of the TCB in the host.

Maintaining the Trust and Isolation

In the current generation of cloud virtualization software, such as XEN [60, 155], there is usually a thin layer of software interacting directly with the hardware, generally referred to as hypervisor. Resource maintenance, coordination, and user controls are implemented in a special guest called the legacy host or Dom0 in XEN terms. The legacy host performs many of the management functions by interacting with the hypervisor. For the rest of the discussion, we will use the XEN terminology, though the same reasoning applies to other virtualization software like KVM [10] and VMware [21]. Traditionally, the isolation between a legacy host and guest VMs is accomplished by the hypervisor. However, under the threat model of TUDEC, the hypervisor is no longer sufficient. We decompose the isolation of computing into processor isolation, memory isolation and peripheral device isolation, which would be examined in detail in the next three subsections.

Maintaining Processor Isolation In TUDEC, each guest has its own dedicated processor(s). This implies direct control of LAPIC. The guests would be capable of sending IPI to processors of the physically collocated VMs. There are two types of IPIs, the maskable ones and non-maskable ones. During the assisted boot up, Global Interrupt Flag (GIF) will have to be cleared so that

any interrupt that attempts to alter the execution on the guest cannot be triggered. After entering the VM mode, the interrupts are enabled so that they can be delivered right into the guest. If a malicious host or guest sent an execution altering the IPI to the guest, triggering a VM exit, TUDEC will be able to examine the exit information. The system can consider it as a malicious attack on the VM and either terminate or ignore the IPI depending on the desired policy. While this does not eliminate the attack, the risk of control flow exploitation is mitigated to a denial of service attack. Advanced attackers with the goal to steal information would not expose themselves by launching DoS attacks, and thus the mitigation is sufficient.

Maintaining Memory Isolation: Each individual guest has her own dedicated physical memory in the host platform during the entire execution. The isolation will hold as long as the adversary cannot access or modify the memory of the user VM. There are two ways of accessing memory: memory manipulation instructions with processor or DMA using an I/O device. We will analyze how to prevent insider and outsider attackers from using these two ways to access memory.

Isolating outsider attacker from user - An outsider adversary has access to a VM that is physically collocated. Since VM runs only in the guest mode after initialization with second level paging enabled [38, 35], the virtualized guest can access only the physical memory assigned to her. Therefore, it is impossible to directly modify the physical memory of either the hypervisor or any neighboring VMs. On the other hand, since each guest is granted the direct access to network devices in TUDEC, the guest can also program an I/O device to perform DMA attack [174]. This however is prevented with IOMMU.

Isolating insider attacker from user - The insider adversary has access to the legacy host, which makes her much more powerful than the outsider attacker discussed above. With access to the legacy host and the hypervisor, she will be able to modify any addressable physical memory except the System Management RAM (SMRAM) area. Therefore, she can modify most of the system wide configurations, making it extremely difficult to defend against. In order to protect the guest memory against this type of adversary, we have to rely on the SMRAM. System Management Mode (SMM) is a special CPU mode that is different from the protected mode and the real address mode. Its main purpose is to perform platform specific system resource management, such as power control. SMM is entered via a system management interrupt (SMI) which could be triggered by both software and hardware. Upon entering the SMM mode, the microprocessor saves its entire state in a separate memory region known as system management memory (SMRAM), and continues to execute the SMI handler that also resides within the SMRAM. When the SMI handler finishes execution, a special instruction RSM is executed to exit the SMM. Within the SMM, all memory protection and interrupts, including the non-maskable ones are disabled. The SMRAM can be locked so that access to the SMRAM outside the SMM is disabled. Once the SMRAM is locked, even a processor in ring 0 will not be able to access it. Therefore, if we map the guest's physical memory in the SMRAM, then the access to it can be prevented, even from a compromised hypervisor. In the AMD architecture, the SMRAM TSEG area can be defined dynamically through two mode specific registers (MSR) SMM_Addr and SMM_Mask, both of which are local to a processor core. SMM_Addr identifies the base address and SMM_Mask determines the range. The protection on this dynamic range of the SMRAM can be locked or unlocked by writing a 64-bit

password onto the SMM_KEY MSR [38]. A technique called memory double view proposed by Azab et al. in SICE [54] can be used to safeguard memory. The SMM_Addr and SMM_Mask of the processor cores on which the legacy host runs are configured so that the SMRAM covers the entire memory region of individual VMs. In this configuration, only the trusted SMI handler can access the guest memory.

The other attack vector is DMA from the I/O device. Similar to an outsider attacker, DMA attacks are blocked by the IOMMU. However, an insider attacker has the root privilege on the platform, so she can also manipulate the IOMMU configurations to invalidate the protection offered by IOMMU. The protection of IOMMU is also part of TUDEC, and will be discussed later.

Maintaining I/O Device Isolation: Since each virtual machine has its own dedicated I/O resource, there is no sharing of peripheral devices. However, the ownership of such I/O devices implies potential DMA attacks. As discussed in memory isolation above, the attack can be thawed by proper configuration of IOMMU. IOMMU is one of the building blocks for securing I/O virtualization [200]. It translates and protects the memory from DMA transfers by peripheral devices. Devices are assigned into a protection domain with a set of I/O page tables defining the allowed memory addresses, similar to MMU. During a DMA transfer, the IOMMU intercepts the access message and checks its cache (IOTLB) for I/O to memory address translation along with the access right. IOMMU is controlled with in-memory tables and memory-mapped registers. The outsider attacker cannot access these control structures due to the protection of second level paging system, but there is nothing preventing the insider attacker. For insider attackers, there are two attack vectors as follow.

Attacks on the configuration tables - There are two main tables in IOMMU configuration: page translation table and domain mapping table. The page translation table defines the memory mapping and its associated access rights. The domain mapping table maps the Bus Device Function (BDF) number of a particular DMA request into a protection domain in the IOMMU. To defend this type of attack, we map the control structures into the SMRAM region, preventing the legacy host from modifying the contents of the table. However, this also prevents the system software from maintaining these tables, and thus any modifications will need to happen in the SMM. Traditionally, DMA mappings are dynamic and on demand. Every time there is a new DMA request, a new translation entry will be added to the IOMMU. However such frequent switches to the SMM mode could be highly undesirable from both the performance and security perspectives. In TUDEC, we propose to use a persistent DMA mapping strategy. Drivers will reuse a fix setup of memory pages for DMA. Once added, the mapping will stay for the lifetime of the guest execution. This strategy makes IOMMU manipulation a fixed cost during the initialization of a new trusted isolated environment.

Attacks on the configuration registers - Another type of attack attempts to alter the configuration registers of IOMMU, such as the base address register (BAR) of the IOMMU device, which holds the pointers to the memory addresses of the IOMMU control structures. Even though an insider adversary cannot directly modify in memory tables, she can still modify these registers to point to a completely new table elsewhere in the memory. The configuration register is located in the PCI

configuration space of the device. For the AMD IOMMU [28], the base address pointer to MMIO is stored in two registers inside the IOMMU capability block: the base address low register and base address high register. An insider attacker has full control over the PCI configuration space. In TUDEC, we use the *lock once* feature of these two registers. Once locked, they cannot be changed until the next system reset. The BAR is set up and locked during the trusted boot process of the server platform, effectively thawing the attack.

4.4.2 High Performance I/O

I/O virtualization has been widely recognized as the bottleneck of virtual machine execution [136, 104, 49]. Reducing the I/O subsystem overhead has been one of the major areas in the field of virtualization research [60, 155, 104, 136]. Some of the previous works focused either on optimizing paravirtualization of the I/O devices [60, 155] or optimizing PCI-passthrough [104, 136]. However, each guest virtual machine has direct access to its own physical resources in TUDEC. With direct access to the processor core and its LAPIC, the guest can receive and acknowledge interrupts from I/O devices without trapping out to the hypervisor. This is considerably more efficient than the current implementation of PCI-passthrough in the hypervisor, which still traps and emulates all the interrupt deliveries and completions [60, 155].

One of the reasons for the hypervisor involvement is the difference in the interrupt vector between the host system and the guest system. In traditional platforms, the operating system sets up the PCI device with an interrupt by choosing an available vector and placing the interrupt handler in the corresponding position in the interrupt descriptor table (IDT). However, the guest operating system does not have any information about the underlying hardware platforms to pick the appropriate interrupt. In order to eliminate this indirection, the host system will have to first search for a suitable vector and then pass it on to the guest as a configuration parameter during the start up. The guest will then use this interrupt vector assigned by the host. Furthermore, under a persistent DMA mapping strategy, DMA request can reuse a set of pre-configured pages with existing IOMMU mappings. This strategy eliminates the time needed for manipulating IOMMU mapping for each DMA request and further improves the I/O performance.

From a security perspective, an outsider adversary can launch a DoS attack on the server by either handling every interrupt including the ones for the hosts and other VMs or holding onto an interrupt line for a long time. In order to mitigate this attack, we utilize IOAPIC interrupt affinity mapping to deliver only interrupts belonging to the VM. This is possible because any system-wide interrupt vector for the guest is a result of negotiation of available vectors decided by the host before the VM starts.

4.4.3 Data at Rest

In TUDEC, any sensitive user content will be encrypted while at rest. Files are only decrypted within the container for computation, and will be encrypted before leaving the container to deliver via the bus to any I/O devices. Therefore, the confidentiality of data at rest is guaranteed.

4.5 TUDEC Operating Architecture

In this section, the operational aspects of the TUDEC system are introduced. Instead of focusing on the design principles and rationales, we will emphasize on the integration, deployment, and operation sequence of TUDEC. The systematic components of TUDEC will be presented first, followed by the temporal aspect of the system. There are two interesting subsystems. The first subsystem is the startup, execution, and shutdown of the hosting server. The second subsystem is the life cycle of individual guest virtual machine. The hosting server will start up and remain in the execution state for extended period of time until some schedule event. Meanwhile, virtual machines will be started, paused, and shut down more frequently.

4.5.1 TUDEC Components and System Integration

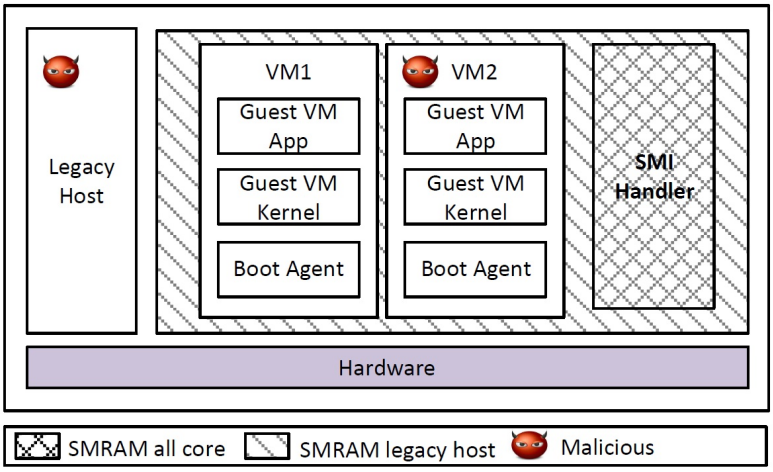

Figure 4.1: TUDEC Architecture

TUDEC, as part of the trusted computing base, is integrated into the server system. There are several components of contributing to different capabilities of the system.

- *Trusted boot* - This component is integrated into the trusted boot code of the hosting platform. It is responsible for the generation of initial content such as communication keys as well as hardware configuration initialization such as locking the SMRAM by setting the D_LOCK bit to prevent potential unauthorized access.
- *Specialized SMI handler* - When a user makes a request for an isolated execution environment, a trusted entity will assist in the remote attestation protocol. This trust anchor in the host is realized by a specialized SMI handler along with the TPM.
- *Boot agent* - The component will act as a temporary ”hypervisor” to assist secure virtual machine to boot up in a contained environment.

Fig. 4.1 shows a typical deployment view of TUDEC. The legacy host is the entity that coordinates resource management. Applications of the user run on top of the guest operating system (system image). Boot agent is a hypervisor-like entity that will assist with the trusted virtual machine initialization. The specialized SMI handler, which will assist in bootstrapping trust in the execution environment, resides in the SMRAM that is protected from all processor cores. Each individual-trusted execution environment contains a virtual machine, requested by the user, running directly on the hardware after trusted startup.

4.5.2 Host System Life Cycle

When the host server starts up, it will perform a secure boot of the system with the TPM chip. As part of the system initialization, it will generate a public/private key pair which will later be used to facilitate secure communication between TUDEC and the remote attester. The customized SMI handler, which is developed as part of the TUDEC, will be copied into SMRAM and inserted into the SMI handler table. Both the generated key pair and specialized SMI handler are measured and extended onto the TPM's PCR for later attestation. Immediately after the measurement, the SMRAM of the system is locked to prevent any changes, even by the most privileged entity such as the hypervisor, in order to protect the contents within. Lastly, in order to prevent the malicious host from modifying the IOMMU configuration registers, we mapped the control structures of IOMMU into the T_SEG region of the host. The address register that stores the location of the control structure is also locked to prevent further redirection.

4.5.3 Life Cycle of User Virtual Machine

A detailed execution flow of a user VM life cycle is shown in Fig. 4.2, for which we will give detailed descriptions

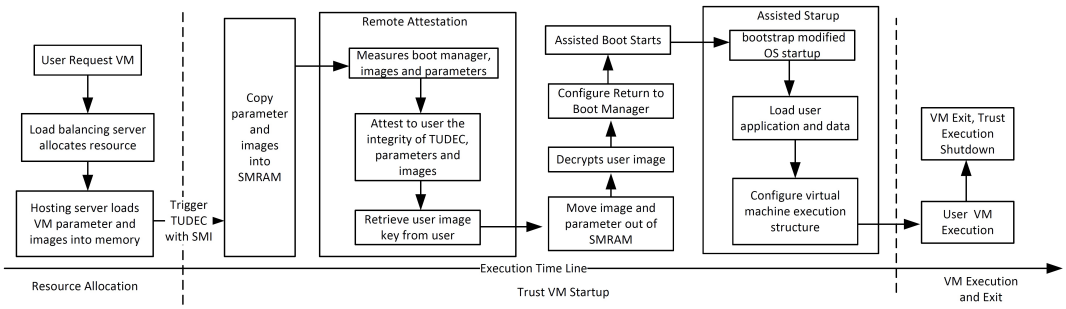

Figure 4.2: TUDEC execution flow from initialization to exit

Resource Allocation

When a user requests a virtual machine, the load-balancing server will communicate with an available hosting server to allocate the physical resources including the physical processor cores, memory ranges, and I/O devices. At the same time, the host will also pick an interrupt vector that is available in the system to let the guest use for direct interrupt delivery of the I/O device. The vector is passed to the guest as a parameter. Once the resources are allocated, the legacy host will request the system image and the encrypted user image from the image server. The system image is shared among all users. Since the user image contains user-owned modules and files, they are unique for each individual user. While the images are being transferred, the legacy host will also load the boot agent, which will assist system startup of the VM image, into memory. Once both images are received and loaded into memory, the legacy host will save all the environment parameters and trigger an SMI to invoke TUDEC. Once in SMM mode, the processors cores allocated to the user VM will start preparation while the other cores will resume normal operation. For the allocated cores, the parameters and runtime images are copied into SMM memory for the integrity verification in the trusted startup phase.

Trusted Startup of User VM

Trusted startup of a user VM consists of three main steps: the remote attestation of integrity, virtual machine resource initialization, and the assisted start up for hardware-only virtualization.

Remote Attestation: There are three goals in the remote attestation step: integrity verification of TUDEC, a secure communication between user and VM, and the integrity verification of images. The host will use standard remote attestation procedure to attest both the integrity of the TUDEC platform and the communication keys to the remote user. The remote user can then use this communication key to establish a secure channel with the host in order to transmit the user image decryption key only if the integrity of TUDEC was successfully verified. Upon receiving the key, TUDEC will move the images into the intended memory region and decrypt the user image.

Resource Initialization: In this step, all hardware resources allocated to the virtual machine will be

initialized to the desired configuration. In particular, the mapping in IOMMU will be established and updated while in SMM mode. As described earlier, the guest will be using a persistent DMA strategy in the TUDEC. Once all the resources are initialized, the execution will move on to the assisted startup.

Assisted Startup: In this step, the guest virtual machine will be initialized with the assistance of the boot agent, who will act temporarily as a traditional hypervisor. Processor and device discovery is one of the very first steps performed by an operating system in a system startup. Hardware configuration of the virtual machine is usually not the same as the physical platform on which it runs. As a result, during this process, the hypervisor-like boot agent will have to perform filtering and emulation so that the user virtual machine will be initialized with the designated resources. More specifically, the boot agent will have to assist in device discovery such as processor capability and PCI device enumeration. When initializing the network interface, the guest will set up the network card using the interrupt vector passed in by the legacy host. On the other hand, the boot agent will also configure the IOAPIC to forward the interrupts onto the dedicated cores for the guest to prevent other malicious guests with direct access to LAPIC to handle the interrupts of the guest.

User VM Execution and Exit

Once the guest system is bootstrapped, it is ready to transition into running mode. For both security and performance reasons, we want the guest to exit as little as possible. In TUDEC, guest operating systems are modified so that the results of sensitive instructions such as CPUID are cached. This will eliminate the VM exits, thus reducing the attack interface of the hypervisor. Once the user image finishes loading, the boot agent will then set up the execution environment by configuring the appropriate hardware virtualization structures and then transitioning the execution into virtual running mode with this structure.

At this point, the control is handed off to user applications. When the guest shuts down or attempts any malicious activity, there will be a VM exit causing the execution context to fall back to the boot agent. The boot agent can then decide what to do with the exit. One simple policy could be to destroy the VM and scrub all memory if there is any VM exit, because an exit is a good indicator of an attack.

4.5.4 Deployment

TUDEC relies on several hardware features to protect the VM. Even though the security attributes of these features have been employed in several research work and prototypes [54, 181, 123], careful examination of the hardware platform is still needed prior to deployment due to the use of multiple hardware features from different components in the system. Part of TUDEC is in either the BIOS or UEFI firmware. An insider attacker can flush the BIOS ROM if it is not locked.

However, if the BIOS is locked by a hardware mechanism, then it would be impossible to push an update to TUDEC remotely. For the future, TUDEC can be built into UEFI [20] as a signed driver.

4.6 Prototype Implementation and Performance

Prototype

We've implemented a TUDEC prototype by modifying the open source hypervisor XEN [60, 155] ver 4.1.2 running on Ubuntu 12.04 server with 3.0.23 Linux kernel. Some of the later updates of XEN 4.2.1 were also ported back to XEN 4.1.2 to support device emulation, since we are interested in the performance difference between an emulated network card, a paravirtualized network card, and direct assignment in TUDEC. The resource allocation routine of XEN is modified to allow dedicated resource assigning. For the guest, we are using a developmental copy of Ubuntu 12.04 server, with 3.2.0 Linux kernel. We modified the kernel to accept a hardcoded interrupt vector for the network card, so that there is no negotiation during the direct interrupt delivery. The guest kernel is also modified to cache sensitive instruction results such that VM exits can be reduced. We set up the guest to boot via iPXE with images from a storage server. The motherboard we picked for the experiment is not supported by the coreboot [7], and the main goal of our experiment is to demonstrate the I/O performance while in the guest mode, so we did not reverse engineered the BIOS ROM to unlock the SMI handler.

Experiment Design and Setup

We are interested in quantifying the I/O performance improvement of TUDEC. Out of all the I/O devices, network interface is one of the most important in the cloud environment. Lastly, network interface is a mandatory I/O device in TUDEC, with direct impact on the performance of the isolated environment. On the other hand, though data intensive executions often have high processor usage, we collected the network performance measurement varying processor utilization. More specifically, one is measured when the processor is idle while the other one is measured when the processor is fully occupied. By conducting these experiments, we can compare the bandwidth among emulated NIC, paravirtualized NIC, PCI passthrough NIC, and TUDEC NIC under different processor load.

Our prototype is evaluated on a desktop machine running AMD FX 8120 processor with 16 GB of memory on M4 A97 R2.0 Asus motherboard. The system is running modified XEN hypervisor 4.1.2 as described. We dedicate only one core and 2GB memory to the legacy host via the grub kernel command line. One processor core and 1GB memory is allocated for the guest. We consider this configuration sufficient for testing the network interface. A HP Pavillion DM1 netbook running Ubuntu 13.04 with Linux 3.8.0-19 Linux kernel is used as the other end point for our network performance experiments. The hosting server, the storage server and the client netbook

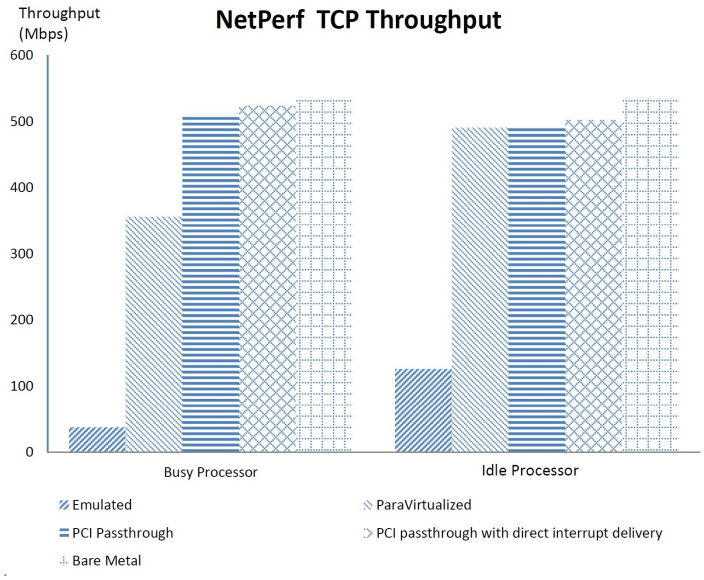

Figure 4.3: NetPerf Throughput Experiment

are connected to a Trendnet TEG-S5g gigabit switch. The storage server is not actively using the network interface when we take the measurement of the network performance. With a minimum number of connections on the switch, we tried to reduce the noise in network measurement.

We use *netperf* to obtain the network performance measure. In the netperf experiment, we run the TCP performance test with 16384 bytes send message size, the max message size possible. As discussed extensively in the previous sections, each packet will accompany varying degrees of VM exits and emulations. With such a large size packet, we are measuring the lower bound of differences in network performance, because a large packet implies fewer exits and emulations.

Performance Analysis

As shown in Fig. 4.3, we perform the experiment under two settings, i.e. in a fully loaded processor and in an idle processor. We load the processor with multiple instances of random number generator to occupy the CPU in the first setting. In the second setting, Dom0 has as much processing power as it needs to perform paravirtualization and device emulation. We believe that the first scenario should be an environment closer to reality, due to the fact that many data-intensive applications actually perform complex operations on the data. In the first setting, TUDEC achieved 98% of bare metal performance, compared to 66% for paravirtualization and 7% for device emulation. In the second setting, TUDEC achieved 94% of bare metal performance, while reaching 91% for paravirtualization and 23% for device emulation. The impact of server load on device emulation is the greatest, while there is significantly less impact on TUDEC with direct device assignment. This result is due to the processing power needed for emulating the device or communicating through event channels for paravirtualization in XEN.

As a result, we can conclude that it is necessary to have direct device assignment or dedicated I/O

resources in virtualization for data intensive applications in the cloud.

4.7 Security Analysis

Isolation of Processor

Individual physical processor cores are dedicated to each guest VM, and thus each user execution is separated. A malicious legacy host can still send unmaskable inter-processor interrupts (IPI) to the user VM, which will cause a VM exit. However, the only action that the boot agent performs upon the exit is shutting down. Therefore, there is no way for the legacy host to alter the control flow of user VM by issuing interrupts.

Isolation of Memory

Isolation of memory is achieved through two mechanisms. The memory of a user VM is isolated from the malicious legacy host by SMRAM protection. Even though there are attacks on SMRAM based on cache poisoning [202], the mitigation [54] is fairly straightforward. This memory is also isolated from neighboring VMs via extended page table, which is set up by the boot agent and requires privilege instructions to change, yet any VM exit will trigger shutdown of the virtual machine.

Isolation of I/O Device

With hardware-based virtualization of the I/O devices, each of these devices will appear as if it is a real physical hardware to the software system. As a result, the isolation of the I/O device is achieved by assigning a different physical or hardware-virtualized device to individual VMs. Also, one should not overlook the fact that each VM has full access of LAPIC, though the risk of malicious guest launching DoS attack is mitigated by delivering interrupts onto the devices that generate them. The host can actually program the IOAPIC registers to change such delivery. However, even though the host can change the destination of the interrupt delivery, he can neither change the execution flow of guest nor access memory of the guest. Therefore, the most she can accomplish is to take over all the interrupts and launch a DoS attacker, but the insider adversary is not interested in such attack as defined in the threat model. An adversary can still try to congest the I/O bus by flooding it with the messages. However, the effect has been experimentally proven to be quite insignificant [181].

Data at Rest

For many applications, it might not be desirable to make the executable public, because malicious attackers can perform various binary analyses on it. Therefore, in TUDEC, we aim to provide data-at-rest security service by encrypting the user image with user-maintained secret key. During execution, the user will provide the key only if the attestation on the isolated execution is successful. Hence, the key only exists in the trusted isolated domain protected by TUDEC. Furthermore, the key for decrypting this application image is transmitted during the attestation using a secure channel so that the confidentiality can be guaranteed. The management of user keys can be accomplished using existing key infrastructures and is not in the scope of TUDEC.

Availability

TUDEC could not defend against DoS attacks from an insider adversary. She could simply remotely shutdown various servers in the cloud to deny access. However, it could defend against malicious neighboring VM quite effectively, since all of the hardware resources are dedicated except the bus. Contention attacks on the bus were demonstrated in [181] to have little effect.

4.8 Related Works

Data Security in the Cloud

Data security and privacy has been one of the major focuses on cloud security. However, most works assume that data has to be encrypted before outsourcing to the cloud. Therefore, various cryptography-based schemes have been developed to operate on the cipertext in order to ensure user data privacy, confidentialty, and integrity, etc. [192, 75, 205, 179]. This line of research takes a totally different approach. TUDEC aims to provide a secure environment to operate on data in plaintext form while they are being processing in the cloud.

Securing the Hypervisor

For hypervisor security, there are generally three approaches to providing isolation of individual VM in the cloud environment: hypervisor-based approach [126, 173, 142, 160, 169, 97, 196], hardware-assisted isolated execution environment [144, 54, 178], and using direct hardware access to provide necessary isolation needed [123, 181]. Within the hypervisor-based approach, research generally focuses on either minimizing hypervisor [126, 173, 142, 160, 169] or hardening it [196, 53]. In hardware-assisted isolated execution environment [144, 54, 178], systems use hardware functions to maintain the isolation. Our work is inspired by SICE [54], in which Azab et al. proposed to use SMM memory to isolate individual workloads from potentially malicious

hypervisors. However, the system still relies on the correctness of security manager and device emulation in the legacy host. Furthermore, the performance drawback of device emulation is not tolerable for data intensive applications TUDEC aims to support. The last type relies completely on the isolation provided by the hardware [123, 181]. TUDEC has a different trust assumption on the legacy host.

I/O devices Virtualization Performance Optimization

Reducing the I/O subsystem overhead has been one of the major area of focus in the field of virtualization research [60, 155, 104, 136]. However much of the previous work has focused on optimizing paravirtualization of the I/O devices [60, 155]. ELI [104] is an exit-less interrupt delivery system proposed by Gordon et al, which has similar goal to our system, except their assumption of unmodified guest and no dedicated hardware. TUDEC takes advantage of direct physical resource assignment.

4.9 Summary

In this work, we've presented TUDEC, a trusted execution environment optimized to provide close-to-bare metal performance for data-intensive execution in the cloud. Instead of performing computation on encrypted data, which has been recognized as prohibitively expensive, we propose to perform computation on decrypted user data inside a trusted execution environment. This isolated environment is capable of safeguarding user data even when the software on the neighboring virtual machines or host is compromised. We've built a prototype and have demonstrated that the TUDEC has significant improvements over device emulation in the previous trusted isolated environment, thus more suitable for data-intensive application. In the future, we plan to further investigate how to apply TUDEC to different platforms and integrate into the system as an UEFI module.

Chapter 5

Defense - CaSE : Cache-Assisted Secure Execution on ARM Processors

5.1 Introduction

Smart devices are playing an increasingly important role in our daily life. As the most widely deployed CPU in mobile devices, ARM family processors have been used in 4.5 billion mobile phones to process and store sensitive data [41, 40]. For instance, around 51% of U.S. adults bank online and 35% of them use mobile phones to perform online transactions [93]. Meanwhile, fueled by the lucrative black market for mobile malware, an increasing number of system vulnerabilities have been identified and exploited to compromise the mobile OS [188]. McAfee Lab reported a 24% increase in the unique number of mobile malware in Q4 2015 [44].

To enhance the security of embedded systems, ARM provides a hardware security extension named *TrustZone* to protect sensitive code and data of applications in an isolated execution environment against a potentially compromised OS [27]. TrustZone has been widely adopted not only in academic research projects [140, 212, 132, 121, 201, 165], but also in commercial products [17, 15, 52]. However, the design of TrustZone cannot prevent physical memory disclosure attacks such as cold boot attacks [111, 147, 77, 102]. Since mobile phones are frequently stolen, when attackers have physical access to the mobile devices, they can gain unrestricted access to the contents in the DRAM. Unfortunately, TrustZone does not enforce encryption of memory in the privileged environment like SGX [48, 61]. As a result, sensitive information, such as cryptographic key material, is not secured even if it is stored in TrustZone protected physical memory when adversaries have physical access to the mobile device.

To protect against physical memory disclosure attacks, SoC-bound execution solutions have been proposed to move sensitive data out of DRAM and save them in processor registers [146, 170, 105], processor cache [152, 108, 187, 77] or internal RAM [77]. All these SoC-bound execution solutions can effectively thaw physical memory attacks under a strong assumption that the OS,

which is responsible for creating and maintaining the SoC-bound execution environment, can be trusted. The justification for this design assumption is that when the OS is compromised, there is no need for attackers to launch a cold boot attack, because the OS can directly access the entire DRAM. However, it is not true for ARM processors with TrustZone support. Though TrustZone can prevent a malicious OS from accessing protected secure memory, it cannot defend against cold boot attacks. Thus, it is critical to protect mobile systems against multi-vector attacks [56] including software attacks and physical memory disclosure attacks.

In this work, we propose a cache-assisted secure execution system called *CaSE* that can protect against both software attacks and physical memory disclosure attacks on ARM-based devices. The basic idea is to create a secure environment in the CPU cache and use TrustZone to prevent the potentially compromised OS from accessing the secure environment. Thus, CaSE can protect both confidentiality and integrity of the application's code and data against both software attacks and physical memory disclosure attacks.

To protect against physical memory disclosure attacks, CaSE creates an execution environment inside the ARM processor by loading and executing an application completely within the CPU cache. Cache is designed to be a hardware mechanism that is transparent to the system software except for a small number of maintenance instructions. Therefore, we solve several challenges to create a cache-assisted execution environment.

First, to make computation SoC-bound, the application code, data, stack and heap have to be stored in and only in the cache. The memory for each component in the application address space has to be allocated carefully to eliminate cache contention. Unfortunately, none of the publicly available ARM documents details the mapping from memory addresses to cache line indexes. In order to correctly place and optimize application memory in the cache, we design and perform experiments to obtain cache mapping schemes of the targeted hardware platform.

Second, once the application is loaded in the cache, we make use of the hardware-assisted cache locking function to pin down portions of the cache, without significantly impacting the system performance. With the ability to control eviction policy on cache lines that store the sensitive data, it is possible to enable context switching between the protected application and the rest of the system without concerning the execution of other programs will cause eviction of the sensitive contents from cache to DRAM.

Third, since the application is still encrypted when loaded into DRAM, it needs to be decrypted completely within cache before being executed. In many processor architectures, including ARM, instruction cache and data cache are not guaranteed to be coherent. When an application decrypts its own code back into the process address space, instruction cache and data cache become incoherent. Such issue of incoherent cache caused by self-modifying programs is often resolved by flushing the cache. In CaSE, flushing the cache fails our efforts of running applications entirely inside the SoC. To solve this problem, we synchronize the incoherent data cache and instruction cache by utilizing the unified last level cache in the processor.

TrustZone is used to protect the cache-assisted isolation environment against an untrusted OS.

Cache lines in TrustZone-enabled ARM processors are built with an extra *non-secure (NS)* bit to indicate whether the line belongs to the secure world or the normal world. Therefore, the rich OS in the normal world cannot access or manipulate the cache lines used by the secure world. The secret key to decrypt the application is saved in the secure world cache. Without the key, a compromised rich OS cannot decrypt the application code, which may be misused by attackers to reverse engineer proprietary algorithms or find potential vulnerabilities. CaSE offers two running modes depending on whether secure world cache or normal world cache is used to create the environment for the SoC-bound execution. These two modes provide a trade-off between the system security and the run-time performance.

We implement a prototype of CaSE on the i.MX53 running ARM Cortex-A8 processor. Using the CaSE, we show that it is possible to execute a kernel integrity checker and a suite of cryptographic algorithms including AES, RSA, and SHA1 in the cache with small performance impacts.

In summary, we make the following contributions,

- We propose a secure cache-assisted SoC-bound execution framework that can protect sensitive code and data of applications against both software attacks from a compromised rich OS and physical memory disclosure attacks that can gain unrestricted access to the DRAM.
- We present a systematic study on designing and securing our cache-assisted SoC-bound execution environment on ARM platforms. We demonstrate the applicability of our system by prototyping several popular cryptographic algorithms along with a kernel integrity checker.
- We implement a prototype on the i.MX53 running ARM Cortex-A8 processors. The experimental results show that CaSE has small impacts on the system performance.

5.2 Background

We first introduce the ARM TrustZone hardware security extension. Then we discuss the generic ARM cache architecture along with the changes in the cache design due to the addition of TrustZone.

5.2.1 ARM TrustZone

TrustZone is a set of hardware security extensions, consisting of modifications to the processor, memory, and peripherals [27]. It has been supported since ARMv6, and most of the recent ARM system-on-chip processors support this security extension. The main purpose of TrustZone is to provide an end-to-end, complete system isolation for secure code execution. The isolated environment provided by TrustZone is often referred to as the *secure world*, while the traditional

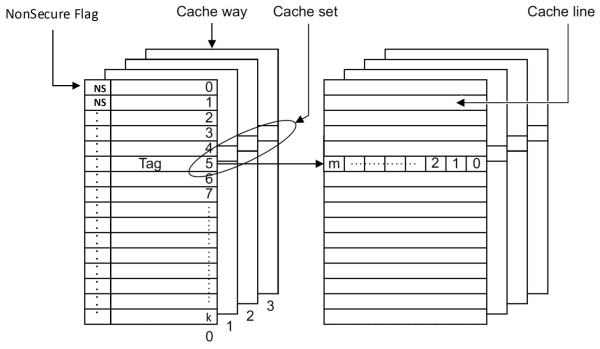

Figure 5.1: Cache Architecture in ARM TrustZone

operational environment is often referred to as the *normal world*, the *non-secure world*, or the *rich OS*.

Based on the world that the processor is in, different system resources can be accessed. The *security configuration register (SCR)* in the *CP15* coprocessor is one of the registers that can only be accessed while the processor is in the secure world. SCR contains an *NS (non-secure)* bit that governs the security context of the processor. When *NS* bit is cleared, the processor is in the secure world. When *NS* bit is set, the processor is in the normal world. The only exception is when the processor is in the monitor mode, which can be triggered by either interrupts or *secure monitor call (SMC)* instruction.

5.2.2 Cache Architecture in ARM Processors

Cache is considered to be the first level memory system in ARM. It is usually constructed with a fast and expensive static random access memory (SRAM). Most of the current processors have multiple levels of cache, including level one (L1) cache, level two cache (L2), and translation lookaside buffer (TLB). Modern high-end processors typically have 32 KB to 1 MB L1 cache, and the size of L2 ranges from 256 KB to 2 MB. Since cache is small compared to the total amount of addressable memory, N-way set associative table is often used to organize the cache.

A typical 4-way set associative table is shown in Figure. 5.1. A physical memory address is indexed into k cache lines, where k is the set size. As there are four tables of size k, the way number is four. Therefore, for any given memory address, it can be in k^{th} set entry of any way. For each cache line, there is a tag carrying the hash value of the index along with the status bits. With the introduction of TrustZone in the ARM architecture, all levels of cache have been extended with an additional *NS* tag bit, which records the security state of the transaction that accesses the memory [27]. It eliminates the need for a cache flush when switching between the two worlds, significantly improving the system performance. The content of the caches, with regard to the security state, is dynamic. Any cache line can be evicted to make space for new data, regardless of its security state. In other words, it is possible for a secure cache line fill to evict a non-secure

cache line, and vice versa.

5.3 Threat Model and Assumptions

5.3.1 Threat Model

Sophisticated cyber attacks nowadays involve multi-stage, multi-vector attacks [56]. We assume that attackers can use both software attacks and physical memory disclosure attacks to obtain sensitive information in the DRAM.

Software Attack

Due to the increasing complexity of the mobile OS kernel, attackers can often exploit various kernel vulnerabilities to compromise the mobile OS. Therefore, we assume that successful software attacks can lead to the compromise of the OS and thus gain unrestricted access to not only DRAM but also the CPU cache and registers.

It is well known that an adversary can use direct memory access (DMA) attacks [67] to gain arbitrary access to physical memory on desktop computers through DMA channels such as FireWire, Thunderbolt, and PCI Express. Though DMA ports are not commonly available on current mobile devices and USB ports are not DMA capable, it is still possible for the attacker to misuse built-in DMA capable I/O devices such as LCD controller and storage controller [188]. Therefore, we consider DMA attack as an attack vector available to the compromised OS.

Physical Memory Disclosure Attack

With physical access to the mobile devices, there are many types of physical attacks, and it is hard to anticipate all of them. For example, if the JTAG interface is enabled on production systems, the attacker can connect a JTAG debugger to manipulate system states of the normal world. Fortunately, the secure world is protected from JTAG with the built-in protections from TrustZone. Instead, the attacker can use other advanced hardware to examine SoC internals or change DRAM state [182].

In this work, we focus on physical memory disclosure attacks, such as cold boot attacks [111, 147, 71], which exploit the remanence effect of physical memory to gain unrestricted read access to system memory. In general, there are two types of cold boot attacks: (1) resetting the computer to load a malicious OS from the attacker, and (2) unplugging and placing DRAM chips into another machine controlled by the attacker. Moreover, attackers can use bus snooping attacks [145] to capture the sensitive data when it is being loaded from or written to the DRAM. Note that though TrustZone can be used to protect secure code execution against the compromised rich OS

in the normal world, the DRAM used by the secure world is still vulnerable to a physical memory disclosure attack since no encryption is enforced on the DRAM.

5.3.2 Assumptions

We assume the ARM platform supports the TrustZone hardware security extension. The high assurance boot (HAB) and system isolation between the two worlds provided by TrustZone can be trusted. We assume the secure application running in the secure world can be trusted and will not leak its information deliberately. The attackers can launch various software attacks and physical memory disclosure attacks in order to freely access the sensitive data in DRAM memory. Moreover, after gaining the root privilege in the normal world through software attacks, the attacker can also access the CPU cache and registers of the normal world inside the processor. However, she will not be able to access the processor cache or registers in the secure world due to the protection of TrustZone. We assume that attackers with physical access to the mobile devices cannot utilize sophisticated hardware to access the SoC-bound data in cache or registers. Side channel attacks such as timing and power analysis are out of the scope of this work.

5.4 CaSE Architecture

CaSE is designed to provide a secure and isolated SoC-bound execution using the commodity hardware components running ARM processors. We first present our security goals and then give a system overview which focuses on how these goals are achieved in CaSE.

5.4.1 Security Goals

To protect against both software attacks and physical memory disclosure attacks, we design CaSE to satisfy the following security goals:

SoC-bound Execution Environment

The computation and memory of the application shall be within the physical boundary of the SoC. Since physical memory disclosure attacks are capable of revealing all memory contents outside the SoC, CaSE needs to use the memory that is within the physical boundary of the SoC, such as on-chip memory or processor cache to create a SoC-bound execution environment.

Isolated Execution Environment

The system shall be able to provide an isolated execution environment. In other words, it shall be able to bootstrap and maintain an execution environment that is completely isolated from the compromised mobile OS, including separation for processor, memory, and peripherals. On ARM processors, TrustZone can be used to achieve this goal.

Memory Protection Outside the Execution Environment

To protect both integrity and confidentiality of application code and data, all program information outside the physical boundary of the SoC shall be protected by cryptography. More specifically, code and data of the application shall be encrypted when they are saved into external DRAM due to memory paging, context switch, etc.

5.4.2 CaSE Overview

The overall system architecture is shown in Figure. 5.2. Cold boot attackers can gain unrestricted read access to all external DRAM, including those used by the system as either the secure world memory or the normal world memory. On the other hand, software attacks allow adversaries to access and manipulate memory contents of the normal world. The protected application is encrypted in the DRAM to ensure its confidentiality. When a user invokes an application, the CaSE controller will first load the encrypted application into the L2 unified cache. Then CaSE controller verifies and decrypts the application completely within cache and sets up the execution environment with cached memory. Using the hardware-assisted memory protection by TrustZone, the cache-based execution environment is isolated from software attacks from the rich OS in the normal world. Lastly, the application context is encrypted before written to memory such that sensitive information never leaves the SoC in plain text.

By executing applications only in the cache of an isolated environment provided by TrustZone, CaSE can defend against both software attacks that compromise the OS in the normal world and physical memory disclosure attacks such as cold boot attacks.

5.4.3 Constructing the SoC-bound Execution Environment

SoC-bound execution ensures that the execution of a piece of code is entirely enclosed within the physical boundary of the SoC. More specifically, the code, data, stack, and heap of the application should all be allocated to the CPU cache. Therefore, cold boot attacks cannot read either the program state or the program itself. To enable a SoC-bound execution in CPU cache, we need to solve several key challenges.

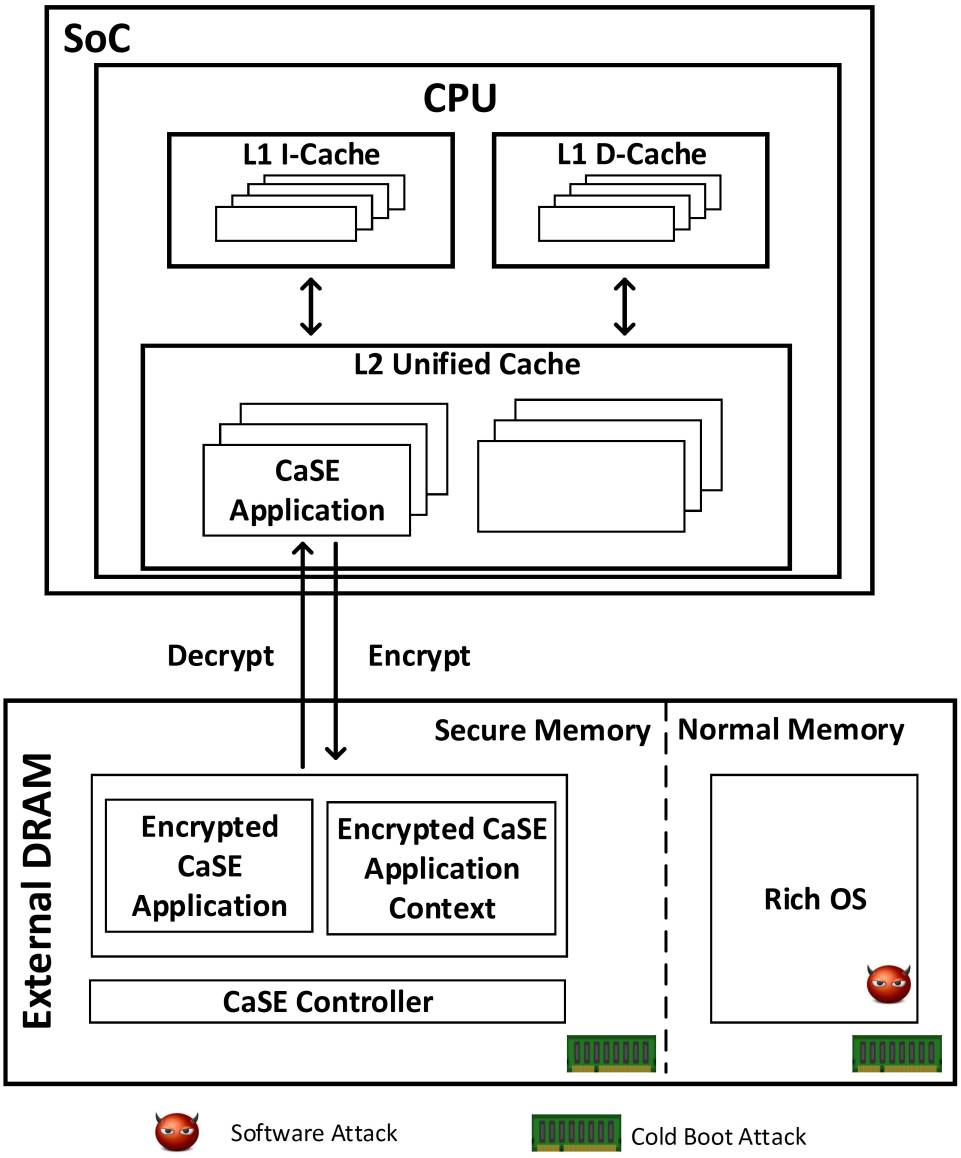

Figure 5.2: System Architecture

First, none of the publicly available ARM documents describes the mapping from physical memory address to cache line index in the cache way. We have to design and perform experiments to figure out this mapping for both L1 and L2 caches in ARM processors. Our results indicate that the cache organization of Cortex-A8 is similar to many other platforms in x86 systems [119, 117]. Second, since there is no direct access to cache lines from system software, we need to develop a method to precisely load memory into cache lines and avoid cache eviction during the load, run and exit stages of the application. Third, when processor cache is used to store both code and data, self-modifying programs can cause cache incoherency between Instruction Cache (I-Cache) and Data

Cache (D-Cache) in the first level cache. We solve this problem by redirecting memory write to the second level unified cache, where the cache lines are used for both instruction and data.

By tackling the three challenges above, our SoC-bound execution can load both code and data into L2 cache and protect the confidentiality of code and data against cold boot attack. However, a compromised OS from software attacks can still access the contents of the CPU cache. Therefore, CaSE also needs to isolate the SoC-bound execution from the compromised OS.

5.4.4 Isolating the SoC-bound Execution from Rich OS

TrustZone provides an *NS* flag in each cache line indicating its security state. Based on the security context of the system, CPU cache is marked as either secure or normal. We call the cache lines used by the secure world *secure cache* and the ones used by the normal world *normal cache*. TrustZone can ensure that the rich OS in the normal world cannot access the secure cache. Thus, a straightforward solution is to use secure cache to create the SoC-bound execution environment, as shown in Figure. 5.2. Alternatively, it is possible to use normal cache to protect sensitive code and data against the rich OS. More details can be found in Section 5.5.1.

5.4.5 Memory Protection Outside the Execution Environment

In CaSE, processor cache is used to create a SoC-bound execution environment. As long as sensitive data resides within this environment, it will remain protected. However, SoC-bound memory, such as cache, is often small in size. When the protected application in CaSE attempts to relinquish resource for other applications, the program context needs to be saved to external DRAM. In order to protect the confidentiality and integrity of these sensitive data, any data leaving the SoC boundary needs to have a checksum, which is then encrypted along with the data. When this data is loaded back in the SoC environment, it is decrypted within the SoC and the integrity is verified with the checksum.

Due to the lack of hardware support automatic encryption/decryption like Intel SGX [48], the cryptographic protection for memory has to be provided by CaSE.

5.5 Design and Implementation

In this section, our design and implementation of the CaSE architecture on the i.MX53 platform is presented. Two execution modes using two TrustZone worlds are first presented. The challenges and our solutions in creating a cache-bound execution environment are then discussed. Next, details on how to isolate the running environment from the rich OS and secure the data outside the SoC are presented. Lastly, the locked cache layout in our prototype and two secure application prototypes of a cryptographic library and a kernel integrity checker in CaSE are discussed.

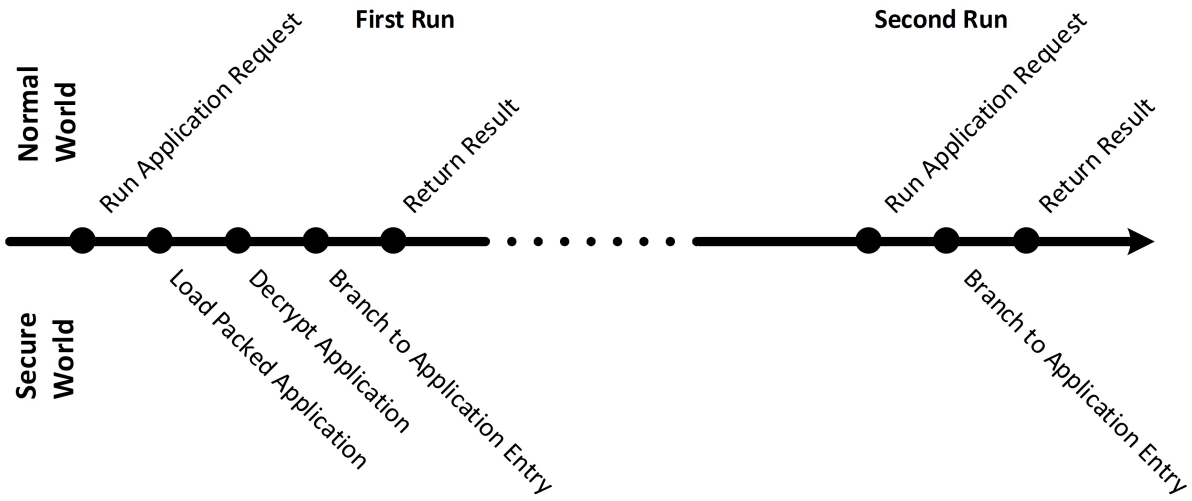

Figure 5.3: Execution Flow using Secure Cache

5.5.1 Two SoC-bound Execution Modes

SoC-bound execution can be performed in either the secure world or the normal world, and these two execution modes offer a trade-off between system security and performance. The overall execution flows of the two modes are introduced in the following.

Execution Flow Using Secure Cache

The CaSE secure mode uses secure cache to create the SoC-bound execution environment. As shown in Figure. 5.3, when a request to run a secure application is received, the CaSE controller loads the encrypted application in the secure cache. After being decrypted completely within the secure cache, the application will run in the secure world until it finishes and sends the results to the normal world. Since the rich OS cannot access secure cache, it is not necessary to clean the application execution environment. Figure. 5.3 shows that in the second run of the same application, the processor can simply branch to the application entry address in the secure cache. Thus, for frequently invoked applications such as cryptographic modules, this property can improve the system performance by eliminating repeated loading and decryption of the application. However, since we run the secure application in the secure world, it will increase the size of the code running in the secure world of the system.

Execution Flow Using Normal Cache

Since the normal cache can be read, flushed or invalidated by the rich OS, it seems difficult, if not impossible, to protect normal cache from a compromised rich OS. CaSE solves this problem

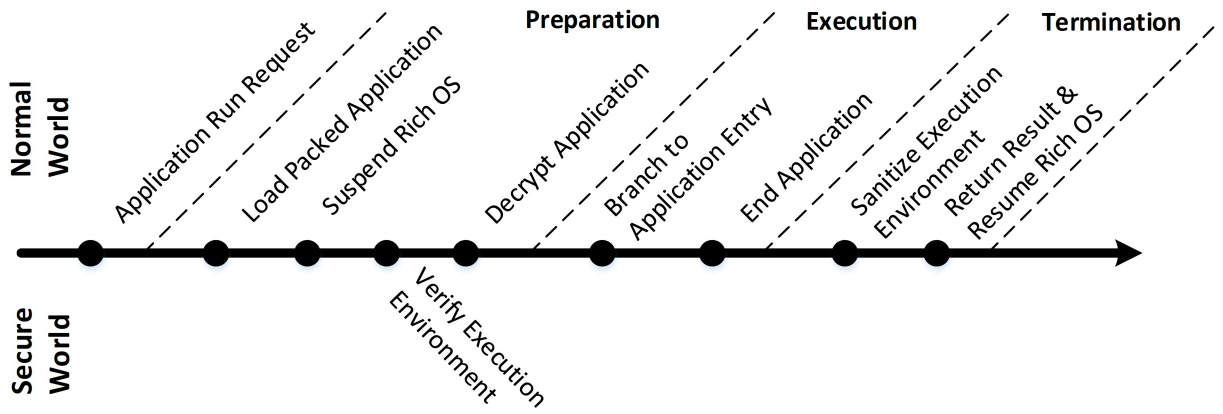

Figure 5.4: Execution Flow using Normal Cache

by relying on temporal separation rather than space separation of the resource. To achieve the temporal isolation between the secure application and the rich OS, we suspend the rich OS when the secure application is running in the normal world.

As shown in Figure. 5.4, when a secure application needs to run, the rich OS will help load the encrypted application into the cache and set up the execution environment in the normal world. After the system switches to the secure world, the rich OS will be suspended. Then the CaSE controller will check the integrity of the application code and its execution environment. If successfully verified, the application payload is decrypted in the cache by an unpacker provided by the CaSE framework. Lastly, control flow will be directed to the application entry function from the unpacker.

For most secure applications such as kernel integrity checking, there is no information that needs to be retained between consecutive executions. However, there are some applications whose states between executions should be kept. Since the application context stored in the CPU cache cannot be protected once the control flow is directed back to the rich OS, we need to encrypt the application context before saving it to the memory.

The benefit of using normal cache is to reduce code running in the secure world since the application runs in the normal world and it cannot compromise the secure world even if it has a vulnerability. When the application execution makes use of the normal world cache, it is necessary to clean the application context including the cache lines and registers before exiting the application and resuming the rich OS.

As a result, the environment needs to be instantiated and torn down each time the same secure application runs, which has an impact on the system performance.

5.5.2 Cache-Assisted SoC-bound Execution on ARM Processor

We tackle three challenges in creating a cache-based SoC-bound execution. First, to optimize the use of the limited cache size, the mapping from memory address to cache lines has to be explored. Unfortunately, neither the architecture document nor the processor document provides details on this translation. We design experiments on real ARM processors to figure out the detailed mappings. The second challenge is to load and lock both code and data of the application along with the execution environment within the physical boundary of the processor. The third challenge is to handle self-modifying programs, particularly, the decryption of the packed application code in the cache.

Reversing Cache Structures

One of the key enablers of our system is the ability to reliably maintain contents in the processor cache. In order to maximize cache utilization, we need to know exactly how the cache controller maps memory addresses to cache set indexes. Only with such knowledge can we precisely and reliably utilize the entire cache. To figure out the mapping from physical addresses to cache set numbers, we first flush both L1 and L2. An *LDR* instruction is used to trigger a cache line fill on the memory location. Once the cache line is filled, we use *STR* instruction to change the values in the cache. Individual cache lines are then invalidated by set and way iteratively. When the set and way being invalidated is the same as the set and way that was used for the cache line fill, the loaded value after invalidation would be different from the value before the invalidation. By repeating this test on different physical memory addresses, we successfully reverse the cache indexing scheme. On ARM Cortex-A8 processors, we conclude that the mapping from memory to cache is linear.

Loading in and Locking down Cache

It is critical in CaSE to load and store the application and its execution environment completely only within the cache. ARM architecture offers the ability to lock down cache entries so that system developers can optimize the cache performance on embedded devices. We utilize this hardware function to lock all the cache lines used by the secure application. The pseudocode for cache locking is shown in Listing 5.1.

The first step of loading memory into cache is to configure the memory address to be cacheable. Memory in the ARM architecture can be categorized into three types, *strongly ordered*, *device*, and *normal*. Furthermore, for normal memory, there are three caching strategy, *write-back*, *write-through*, and *write-allocate*. Write-back and write-through are mutually exclusive. The caching attributes on ARM processors are controlled by various registers, including *system control (SysCtrl) register*, *aux control register*, *L2 lockdown register*, *L2 aux control register*, as well as *page table entry*. The paging table entry controls the caching strategy of the address location using the *type extension (TEX)*, *bufferable (B)*, and *cacheable (C)* bits. The combination of the 4 bits yields

```
1   disable_local_irq();
    enableCaching(memArea);
3   disableCaching(loaderCode);
    disableCaching(loaderStack);
5   invalidate_cache(virtual address of memArea);
    unlockWay(wayToFill);
7   lockWay(allWay XOR wayToFill);
    while(has more to load in memArea)
9       LDR r0, [memArea + i];
    lockWay(wayToFill);
11  unlockWay(allWay XOR wayToFill);
```

Listing 5.1: Lock Memory in Cache

various caching strategies for the memory address location. However, it can be remapped via the *tex remap enable (TRE)* remapping capability in ARM. TRE allows an operating system to have finer granularity control of the types and provides additional room to store OS specific information. When TRE is enabled, memory attributes are mapped to *primary region remap register (PRRR)* and *normal region remap register (NRRR)*.

Since any data written to write-through cacheable memory is directly forwarded to DRAM, we set the cache strategy of the targeted memory area to be write-back, so that memory modifications will be buffered in the cache.

ARM processors provide hardware-assisted cache locking on L2 cache as part of coprocessor functions in CP15. The cache lock register allows system designers to enable and disable the allocation of individual cache way. Once the cache allocation is disabled, the locked cache lines will never be evicted. On the i.MX53, the granularity of L2 cache locking is by individual ways of the cache. Other platforms have different cache locking functionality enabled. For instance, Tegra 3 supports a finer cache locking granularity on each cache line [36].

Using the hardware cache lock is straightforward, but the challenge lies in how to place memory contents in L2 cache instead of L1 cache. This is because we can only lock cache lines in L2 cache. The presence of a cache line in one cache level does not necessarily guarantee its appearance in the other levels. For inclusive cache, any cache line in L1 cache is also in L2 cache. Intel processors largely adopt this inclusive cache paradigm [35]. On the other hand, for exclusive cache, any line in L1 cache is not in L2 cache, and AMD processors usually follow this exclusive cache paradigm [38]. However, for ARM Cortex-A8 processors, there is no indication of inclusiveness or exclusiveness in any document. A closer examination of the cache fill strategy reveals that when there is a cache miss on data or instruction, a cache line fill to both L1 and L2 from the advanced extensible interface (AXI) bus will be triggered. Thus, cache line should be cleaned before executing *LDR* instruction on a memory address.

The code and data that are used in the cache loading have to be configured to the non-cacheable attribute. Otherwise the code itself can be loaded in the cache, causing unintended evictions of the

cache lines that need to be locked. Upon completion of cache filling, the cache way can then be locked. With the cache lines locked in L2 cache, the physical memory addresses corresponding to these cache lines can be used as cache-based memory. In the ARM Cortex A8 processor, L2 cache locking is achieved via the *L2 cache lockdown register* [31].

CPU Bound Application Decryption

One of the system design goals of CaSE is to offer code confidentiality, which is a challenging task. First, the file storage is in the normal world, so the application has to be encrypted while it is saved in the file system. Second, memory contents of neither the normal world nor the secure world are protected from cold boot attacks, so the application has to be encrypted in the DRAM as well. Lastly, the rich OS can be compromised by malware. As a result, the application can only be decrypted either when the rich OS is not running or in the secure world where the rich OS cannot interfere. One of the well-established binary manipulation techniques, code encryption [107, 150], is used to tackle this challenge.

We develop a cache-only packer, *CaSE Packer*, which uses AES to encrypt the code and data of the CaSE application. The entire code to be executed is loaded into cache. The unpacker then decrypts the encrypted code and places them back in the same position inside the *cached* memory, so that the code will remain in cache. CaSE packer, however, has a unique difference from existing application packers on handling the decryption process, due to the cache coherency problems in the ARM platform.

In modern processors [30, 38, 35, 31], the L1 cache (also known as the primary cache) is often split into two parts of equal size, the instruction cache (I-Cache) to speed up executable instruction fetch and data cache (D-Cache) to speed up data fetch and store. The instruction cache is often preloaded with binaries of the executable speculatively using algorithms in the hardware. However, this magic speedup falls apart when application modifies its own code in memory. In many processor architectures including ARM, the I-Cache and D-Cache are not guaranteed to be coherent, and it is up to the system software to handle cache coherency.

During the decryption of the application payload, the ciphertext needs to be loaded into the cache first. This will trigger a cache line fill into the L1 cache. When the ciphertext is decrypted, the results are stored using an *STR* instruction. Due to the close locality between the ciphertext and plaintext, the cache lines that were filled with the encrypted text will be used by the processor to store the decrypted text. Since I-Cache and D-Cache are separated in L1, the *STR* instruction will place the decrypted plaintext code in the L1 D-Cache of the processor. When the control flow is branched to the newly decrypted code, the L1 I-Cache will fetch the instructions from the L2 unified cache instead of the L1 D-Cache because of the cache hierarchy. Therefore, the processor will execute the encrypted code, which will most likely generate an undefined instruction exception.

This problem is often described as the cache coherency issue between I-Cache and D-Cache. The

recommended approach to this problem is to flush out the affected portion or the entire cache to the point of coherence (PoC). Cache contents are written to memory to make sure all masters in the system see the same copy of memory content. Unfortunately, this approach is not suitable for SoC-bound execution, since flushing out contents to memory defeats the purpose of SoC-bound execution. On some platforms, it is also possible to use cache maintenance instruction *clean to point of unification (PoU)* to synchronize the internal caches. PoU is the point by which the instruction and data caches and the translation table walks of that processor are guaranteed to see the same copy of a memory location [30]. The location of PoU is platform and environment dependent. For example, the PoU for Cortex-A15 can be either the L1 data cache or the external memory [39]. Though the location of PoU for ARM Cortex A8 processor is not explicitly documented in the manual, we find through observations in the platform that it is possible to use the clean to PoU instruction to synchronize the internal I-Cache and D-Cache.

Besides using specific instruction, we also devise a method to manually load code into unified cache so that the same method can be used on other platforms where synchronized to PoU instruction is not suitable. Since L2 is a unified cache, synchronization between I-Cache and D-Cache is not needed. We first invalidate the cache lines in L1 for the code memory location. After L1 is cleaned, we use *write-alloc* feature in L2 cache to write to only the unified L2 cache. When write-alloc in the L2 auxiliary control is set, *STR* instruction will trigger only the L2 cache fill. We use this method in CaSE packer to enable the decryption of the application.

5.5.3 Securing Cache-Assisted SoC-bound Execution

It is not an easy task to secure the cache-assisted SoC-bound execution on ARM processors. In the following, some key design efforts to secure the execution environment are presented.

Key Management

Applications are encrypted and packed with a secret key to protect code confidentiality. This key should be stored in the secure storage provided by TrustZone. In our implementation, we make use of the second generation Security Controller (SCC) equipped on the i.MX53 SoC. Using the platform key that is stored in e-Fuse based secure storage on SCC, we encrypt the CaSE master key and store it along with the TrustZone code. When the system boots, the master key is decrypted and stored in the secure cache.

Secure Code Loading in Normal Cache

In order to load the CaSE application in the normal cache, the loading operation should be carried out in the normal world. Since the rich OS may be compromised, we must verify the integrity of the encrypted application code in the secure world. However, according to the TrustZone cache

design, the secure world cannot access the contents in the normal cache. Therefore, it becomes a challenge for the integrity checker in the secure world to verify the integrity of CaSE application in the normal cache.

We use the cache array access feature in the CP15 coprocessor to overcome this difficulty. The cache array function allows a process running in the secure world to retrieve the contents of cache lines whether they are tagged as secure or non-secure. Specifically, we use the *c9* function to read the cache tags and lines into the general purpose registers for inspection. However, since the parameter used in this function is the physical cache array index instead of the cache line index and there is no one-to-one mapping from the array index to the memory location, we have to carry out several experiments to work out the mapping from the physical array index to the cache set and way number. After obtaining the cache tag, we can reconstruct the physical address for each cache line. Then, the memory contents can be used to verify the integrity of the code in the normal cache.

The application stored in the cache is indexed and tagged with physical address, while the processor executes instructions using virtual address. If an attacker inserts a malicious translation from the virtual address to the physical address, she could redirect the control flow of CaSE into any arbitrary physical address [120]. To defend against this memory address translation redirection attack, the translation needs to be locked down in the TLB cache as well.

TLB lockdown function in Cortex-A8 processor is based on the modification of eviction policy [30]. To lock down an entry, the TLB cache for the address has to be cleared out first. The TLB lockdown register is then modified to indicate which TLB line to fill for the next result of translation table walk. To fill the intended address translation at this position, a TLB preload instruction is executed to force hardware to perform a page table walk. Once the cache is filled, the TLB lockdown register is modified again to never evict the entry to achieve the TLB lock. However, similar to the L2 cache, TLB cache is also extended with an extra *NS* bit for the TrustZone architecture. Therefore, if the TLB preload instruction is performed in the secure world, the corresponding translation is for the secure world *only*. To resolve this problem, we fill the TLB in the normal world and then use the TLB data access array function in CP15 coprocessor to verify the translation.

Application Context Sanitization

The CaSE application context consists of the decrypted code, the decrypted data, stack, and heap. All of them are considered sensitive. When the application finishes execution, the context needs to be sanitized. There are two ways to perform the sanitization: overwriting the cache contents or invalidating the cache lines.

When cache overwriting is used, all sensitive cached memory locations are written with a known pattern using *STR* instruction. Then all the cache lines are flushed out to the DRAM such that the changes to the memory is written to DRAM and is no longer cached. When the cache invalidation

method is used, all sensitive cache memory addresses are invalidated using the cache maintenance operation *invalidate by modified virtual address (MVA)*. When the cache memory is invalidated, the cache line is marked as invalid. Thus, all values in the cache line pertaining to the memory address become invalid.

We choose to use the cache invalidation method because it can be used to verify that no sensitive context information is leaked to the memory. To check for cache leaking after the execution of a protected application, we first write pre-defined pattern to the memory location that would be used for the application runtime environment before loading the application. The cache is then flushed to make sure the predefined pattern is in the DRAM. At the end of execution, cache lines used for application execution are invalidated. If there is no cache leak, the result of the *LDR* instruction should return the pre-defined pattern. Otherwise, if the sensitive cache lines had been evicted during the execution of the protected application, then the value will be different.

Handling Cache Coherency between TrustZone Worlds

The cache coherency issue between the normal world and the secure world creates not only the challenge for integrity verification, but also the delivery of computation output. In some application configurations, when the output is cached in the secure world, it is not immediately available to the normal world. For our cryptography library prototypes, the results of encryption that are cached in the secure world are not accessible by the CaSE driver in the normal world until the cache lines are evicted.

To make the output immediately available to users in the normal world, the CaSE application in the secure world has to flush the outputs that are being cached. However, we cannot simply flush the entire cache, since sensitive contents in the cache will also be written to DRAM. There are two methods to solve this problem: clean by MVA or clean by set and way of individual level of cache. When the clean by MVA method is used, CaSE needs to invoke clean by MVA for all the memory addresses of the output buffer. When the clean by set and way method is used, CaSE needs to walk through all the non-sensitive sets in all ways across all cache layers. More specifically, since L1 does not provide locking capability, there is no way to know if a line contains sensitive data or the computation results. Therefore, all the lines in L1 are clean. However, for L2, we know the memory organization of cache ways that are locked. Thus, we can clean all the non-sensitive sets and ways. Our implementation uses the aforementioned method based on the size of the output. When the size of the output is large, it is better to flush the all the non-sensitive cache.

On the i.MX53, *c7 c10* system coprocessor function is invoked with opcode 1 to clean the set and way, and the same function is invoked with opcode 2 to clean by MVA to point of coherency (PoC). Point of coherency is where the processor core and other masters such as DMA controller see the same copy. For i.MX53, PoC is the DDR memory.

Securing Across Power States

An energy-conscious mobile device will switch the processor into different power states to save energy. When the processor is put in the sleep state, power supply to processor cache is cut down, so all data stored in the cache will be erased. This poses a challenge for CaSE, which uses processor cache to create the execution environment.

A simple solution is to keep the cache powered. Both L1 cache and L2 cache can be placed in a different power domain than the integral core. However, this approach has its drawback in power consumption. Modern cache is often constructed with SRAM, which consumes more power than DRAM. An alternative method is to store the cache context in memory that is physically inside the SoC, such as the on-chip RAM (OCRAM). However, many BSPs have claimed the usage of OCRAM for other subsystems [14]. Furthermore, some platforms might not have built-in support to include OCRAM in the secure domain.

In CaSE, we adopt the method that encrypts the cache and saves it in DRAM when the device is in power saving modes. When the system sleeps, the rich OS notifies the CaSE controller to encrypt the cache contents with the master key and then save them into the DRAM. When the system resumes, the contents are loaded back from the DRAM into the cache along with the master key recovered from the secure storage.

5.5.4 Application Development

CPU Cache is one of the key elements to improve system performance in modern processor design. Cache-assisted SoC-bound execution system will inevitably have an impact on the system performance when locking down portions of cache for special usage. Therefore, it is important to optimize the usage of locked cache. We first present the general layout of the locked cache and then our prototypes of two secure applications using CaSE.

Layout of Locked Cache Way

In a typical layout of CaSE application, we place the master key, which is used to decrypt CaSE applications in the first set of the way, followed by the encrypted code and data sections of the CaSE application. CaSE packer code and the environment setup code are immediately after that. Lastly, the rest of the cache lines in the way are used for stack and heap of the application.

There are several essential components for the execution of a binary image, including the libraries, the virtual memory address space layout, code packer/unpacker, and the stack and heap. First, applications cannot use the library provided by the rich OS, since the library integrity cannot be guaranteed. Therefore, CaSE applications need to be statically linked into the binary itself. For our prototypes, we make an effort to modify all the code so that they are self-contained. This is also a byproduct of the effort to minimize the binary code size.

Second, the physical address of the application address space needs to be carefully crafted to fit in a single cache way without causing a collision on the same cache set. We create a set of linker scripts to work with our customized CaSE loader instead of using the default loader and linker script. More specifically, the linker script configures the start address of the binary and the section arrangements.

Third, a CaSE packer is used to encrypt the application binary to provide code confidentiality outside the cache. The packer in our prototype uses AES encryption in CBC mode. Using our own loader simplifies the design of the packer and unpacker, since it is not necessary to implement all the ELF file standards.

Lastly, a custom heap and stack is provided. The stack is allocated by the CaSE loader from the cache memory. Furthermore, we create a simple heap management library, which allocates heap spaces in the cached memory, as well.

Two Secure Applications

Unlike the previous approaches [146, 105, 108] that are designed for a specific algorithm, CaSE offers a generic execution environment. In other words, users do not need in-depth knowledge of the application to create a SoC-bound execution. We develop two secure applications using CaSE application framework.

First, we build a cryptography library by porting AES, RSA, and SHA1 from polarSSL library [57]. Cryptography is one of the fundamental building blocks in modern day computer and network security. Due to the small size of cryptography libraries, it is feasible to place them in the secure cache. The unique advantage of executing in the secure world is the ability to switch context without environment sanitization. As shown later in the experiments, this execution mode has little performance impact on the rich OS, yet offering enhanced security protection. We place AES, RSA and SHA1 all into one library called CaSE crypto library. By combining SHA1 and RSA in the same library, we are able to save some code space due to the use of shared library. Lastly, we need less than one L2 cache way to construct the CaSE cryptography library execution environment.

Second, we build a kernel integrity checker that is invoked periodically to verify the integrity of the rich OS kernel code page. In particular, we calculate a SHA1 checksum of the all the kernel code pages to make sure that it is not modified by any malicious software. Most development efforts are to remove the dependency on the rest of the polarSSL library and the c standard library. We run the kernel integrity checker as normal world SoC-bound execution application, since there could be different implementations of system integrity check and it is difficult to include all variances in the secure code base of the system. Thus, the normal world execution environment is more suitable for the kernel integrity checker.

5.6 Experimental Evaluation

In this section, we first introduce the experiment setup in 5.6.1. The sizes of various system codes are examined in 5.6.2. Cache behavior on the platform is studied in 5.6.3. The last part of the evaluation in 5.6.4 examines the performance of CaSE application system as well as the performance impact of cache locking on the system.

5.6.1 Experiment Setup

We implement our prototype of CaSE on the FreeScale i.MX53 mobile development board. It features a single ARM Cortex-A8 processor with 1GB DDR3 DRAM and 128 KB onboard internal RAM (iRAM) and 16 KB secure iRAM. The system boots with onboard flash along with the uboot and kernel supplied by the Micro-SD card inserted. We use the FreeScale Android 2.3.4 platform with a 2.6.33 Linux kernel. There are two levels of cache in the ARM Cortex-A8 processor. Both the L1 data cache and L1 instruction cache are 4 way 128 set associative cache with 32 KB size. L2 cache is an 8 way 512 set associative cache with 256 KB size. The Android OS is ported from secure domain to the normal domain based on the Board Support Package (BSP) published by Adeneo Embedded [14].

5.6.2 Code Size

The system TCB consists of three components. The first component is the trusted boot code, which is about 500 source line of code (SLOC). The second component is CaSE controller, which is responsible for handling CaSE environment initialization and clean up. It has approximately 500 SLOC of code. The third component is related to specific application implementation. In CaSE, we use SHA1 to check the integrity of isolated execution environment cache, and AES to encrypt application state while it is paused. The SHA1 implementation is 166 SLOC, and the AES is 579 SLOC. In total, there is 745 SLOC for the cryptographic libraries. This additional SLOC does not necessarily have to be included in the TCB if the system only requires secure execution mode.

While SLOC number offers a good estimation of the size of the TCB, it is also important to show the size of the binary code for SoC-bound execution. This gives an idea of the feasibility of fitting the application in the cache. On the Cortex-A8 processor, one L2 cache way is 32 KB.

The code size shown in Table 5.1 is compiled from C and assembly source code using ARM Thumb-II encoding. We turn on the ARM interworking mode during code generation in the compiler. This increases the code size but makes it easier to interwork with the ARM code in our security monitor. With careful coding between function calls, one can remove this compiler flag to further reduce the size of the binary code.

Application	Code+Data (KB)
AES	2.4
RSA	10
SHA1	5
CaSE Crypto Lib	17.4
Kernel Integrity Checker	6.6
CaSE Packer	2.8
Packed CaSE Crypto Lib	20.4
Packed Kernel Checker	9.5

Table 5.1: CaSE Application Size

5.6.3 Cache Bound Verification

Cache-bound verification is designed to verify the security attributes of CaSE. Specifically, there are several aspects of the system we want to verify. First, we need to verify that the location of the CaSE application data indeed exists only in cache, and there is no cache leak during the execution of the program. Second, we want to verify that the processor cache that has been locked down using hardware functions cannot be read or written by the DMA attacks. Third, we examine the interaction of the locked cache lines with cache maintenance operations.

Verifying Applications Exist Only in Cache

The primary goal of this test is to show that the sensitive application is indeed in and only in the processor cache. Furthermore, we want to verify that the CaSE execution environment will not leak any application code or data into the DRAM at any point in time during the execution. Without hardware support, it is difficult, if not impossible, to verify this property for each processor clock cycle the application uses. We choose two points in the execution flow that are likely to show leaked contents if the cache had been flushed to memory. The first point of inspection is at the completion of unpacking action. The second point is when CaSE execution completes, but before the environment is cleaned up.

In this test, we use the packed kernel check application as the test case. We inspect right after unpacking and when the kernel check completes. For both tests, we instrument the code to invoke leakage check routine right after the unpack operation and kernel check. The leakage routine is stored in memory outside CaSE application. The check routine will invalidate all the cache lines occupied by the CaSE application, and then read back the memory location. If the read back value is not the default value for those memory (0xFFFFFFFF), then there is a leak from the protected application. In both points of execution, we detect no leak in the CaSE environment from cache to memory. In order to assure that there is no integer value of 0xFFFFFFFF leaked, we also try

another pattern (0xABABABAB), and the results are the same.

Verifying the Effect of DMA Attack on Cache

One of the main design objectives of CaSE execution is to defend against compromised rich OS. Even though the rich OS is paused during the execution of CaSE application in the normal world, it is still possible for the rich OS to program an I/O device to perform DMA memory read and write to the normal world memory. To see how DMA will interact with cache lines, we program the serial controller to perform DMA read and write to the memory that we lock in cache. More specifically, we load the application in the normal world cache, and use a kernel module in the normal world to program the serial port using DMA to dump memory over serial. We observe that the dump fails to extract contents from cache.

Verifying the Effects of Maintenance Operations on Locked Cache

This test studies the effects of cache maintenance operations on locked cache lines on the i.MX53 and verifies that secure cache cannot be manipulated by cache maintenance operations executed in the normal world.

Common cache maintenance operations include cache clean and cache invalidation. These cache maintenance operations are coprocessor functions that can only be initiated by the CPU itself. Thus, attackers can only launch cache maintenance attack from the rich OS. When one CaSE application is running in the normal cache, the rich OS is suspended, so there is no software attack from the rich OS. However, when the rich OS resumes the system control, it can attempt to use cache maintenance operations to launch attacks on the secure cache, where the master key and the secure CaSE application are stored. In this experiment, we want to verify that malicious code loaded in the rich OS cannot clean or invalidate the secure cache.

We begin the experiment by writing 0xFF to all the memory buffers. We then fill one cache way with normal world cache lines of pattern 0xAB, and another with secure cache lines with pattern 0xBC. Once the two ways are locked, we use the CaSE driver in the normal world to execute the cache maintenance operation. To see if cache clean instruction in the normal world can evict the locked cache lines, we execute clean instruction on all the locked cache lines in both the secure world and the normal world. If the cache contents were written to memory due to the clean instruction, the value read back will be 0xAB for normal world and 0xBC for secure world. In our experiment, memory in the normal world reads back as 0xAB while the memory in the secure world reads back as 0xFF. This verifies that cache clean instruction invoked by the normal world will not be able to affect cache lines in the secure world. We follow a similar procedure for cache invalidation, except that INVD instruction is used instead. We observe that cache lines in the normal world are invalidated, because the read back value is 0xFF, while values read back from the cache lines in secure world remain 0xBC. Therefore, the rich OS cannot use cache maintenance instruction to manipulate the cache lines of the secure world.

5.6.4 SoC-bound Execution Performance

We study the performance of the system by examining the time breakdown of the CaSE application execution. We also compare the performance difference when the secure application is running in the normal cache or the secure cache.

CaSE Isolated Application Performance

As a case study, we use the kernel integrity checker as a normal world application. The packed CaSE kernel integrity check application is 9.5 KB in size. The timing breakdown for each major operation in the CaSE execution is shown in Table 5.2.

Operation	Time (μs)
Environment Preparation	613
Environment Integrity Check	1540
CaSE Unpacking	5973
Kernel Check	18676
Environment Cleanup	412
Total Time	27214

Table 5.2: Kernel Integrity Checker in Normal Cache

From the time breakdown, we can see that though environment setup and cleanup consume some processor cycles, the major computation overhead originates from the unpacking process, which decrypts the encrypted CaSE application payload. The entire kernel check takes 0.02 second to complete, and the application context saving time is 94 μs.

CaSE Secure Application Performance

Using the crypto library as a case study for the CaSE secure execution mode, we measure the benchmarks for a secure cache execution similar to the normal cache execution. Table 5.3 shows the time breakdown of a secure call to perform encryption using AES CBC mode. In the secure mode, the cache is protected against the compromised rich OS. Therefore, it is not necessary to clean up the execution environment.

Performance Trade-off between Execution Modes

To find out the impact of SoC-bound execution environment on application performance, we run AES, RSA, and SHA1 in different environments and compare their performance. First, we port

Operation	Time (μs)
World Switching	2.6
AES CBC Encrypt (1KB)	443
Output Synchronization (1KB)	2
Total Time	447.6

Table 5.3: AES Encryption in Secure Cache

the application into a kernel module and load the module into the rich OS to measure the performance without any security enhancement. Second, we run the application in the two CaSE execution environments, one in the normal world and the other in the secure world. We consider that the first experiment should achieve similar performance as other kernel encryption solutions, and should serve as a good baseline for comparison. On the other hand, the CaSE execution will suffer performance penalty for the enhanced security.

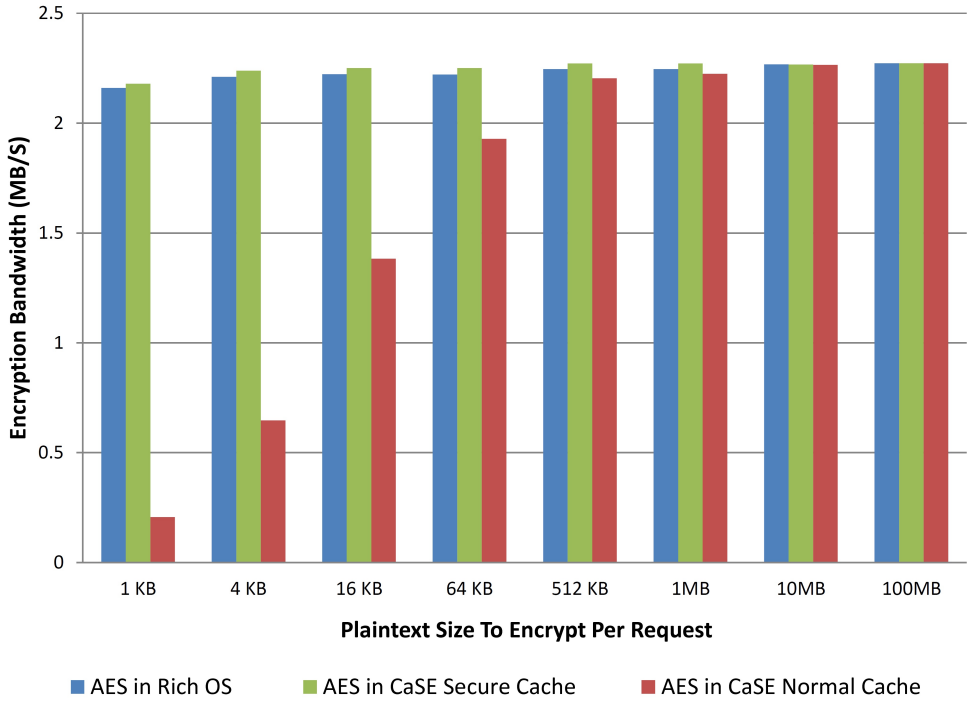

Figure 5.5: AES Speed Comparison

The experimental results on AES algorithm are shown in Figure. 5.5. The performance of secure executed AES is almost identical to that of generic AES. The secure AES has a small advantage over the generic kernel AES when the memory buffer to be encrypted is small. This is due to preloaded cache lines for the AES data section. For smaller size encryption requests, the normal cache execution is significantly slower than the other two methods. This is because the environment is created and destroyed for each request in order to protect the confidentiality and integrity

of the execution environment. However, as the size of the plaintext increases, the difference in the encryption bandwidth diminishes. This is because the overhead to create and destroy the environment becomes insignificant.

We have also performed the same set of experiments on RSA algorithm and SHA1 algorithm. The results for RSA algorithm are shown in Figure. 5.6. In this experiment, we measure the number of 1024-bit RSA decryptions that the system can carry out in one second. Similar to AES, the normal cache execution takes a penalty in the environment initialization and clean up. However, as the number of messages in the request becomes larger, this fixed cost can be ignored. Lastly, we also benchmark the performance of SHA1. We build up our test case by sending fixed size 512 byte packet to the SHA1 module to calculate the hash. Due to simplicity of SHA1, the normal world execution overhead is high when the number of messages per request is low. Similar to RSA and AES, the environment penalty becomes small as the number of messages increases.

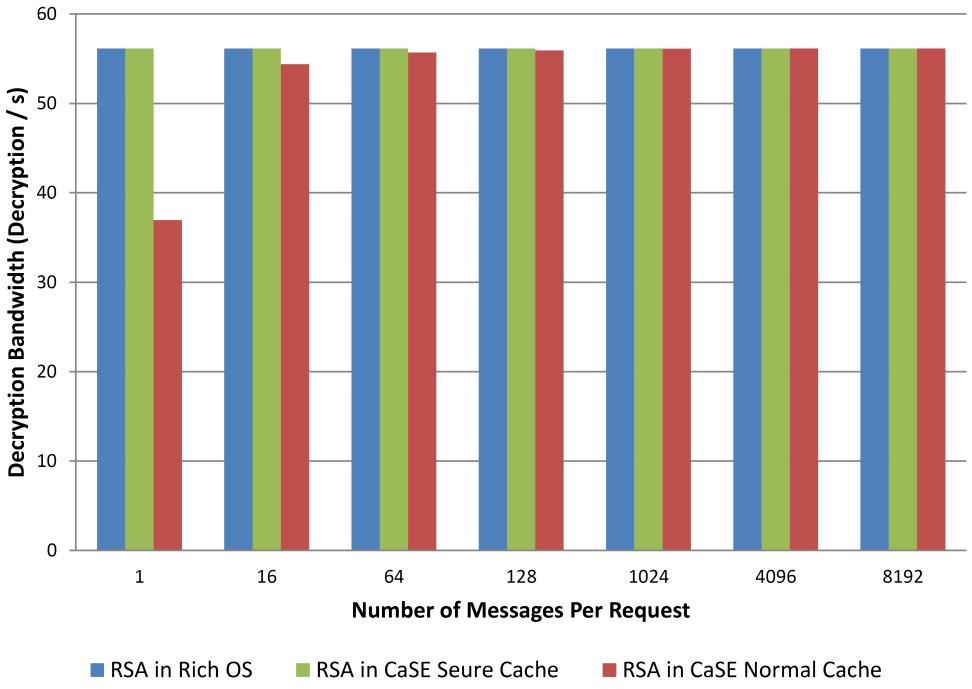

Figure 5.6: Comparison of RSA Operation

Impact of Cache Locking

Cache is originally designed to enhance system performance. When it is locked intentionally for non-performance reason, the system suffers. We use various benchmarking tools to assess the impact of cache locking on the system performance when different portions of cache are locked down. In our implementation, only one way of the L2 cache is locked to reduce the impact on the system.

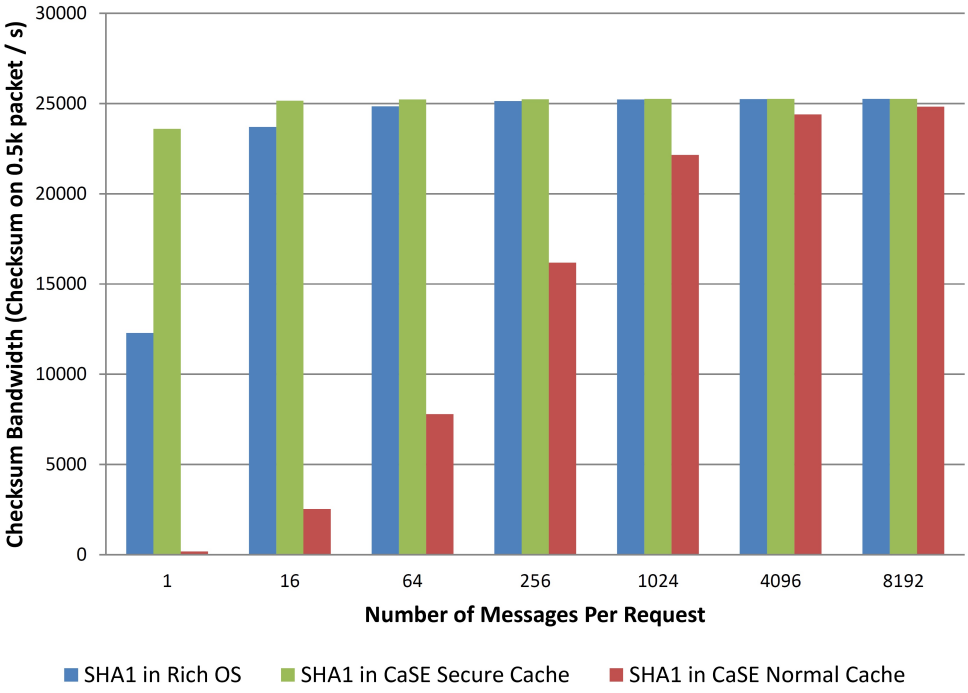

Figure 5.7: Comparison of SHA1 Operation

This experiment is designed to explore the trade-offs between the size of the CaSE application and the impact on system performance due to its monopoly on the L2 cache of the system. We use three benchmarking tools, randspeed [137], linpack [84] and AnTuTu [5]. RandMem measures the performance of random access on large array of memory [137]. The performance benchmark of this tool relates closely to the performance of the memory subsystem. Therefore, with more cache locked away, the system suffers bigger penalty in memory performance. Since Linpack measures integer operation speed of the system [84], the reduction in L2 cache has a smaller impact in LinPack benchmark. Lastly, AnTuTu [5] is a comprehensive benchmark suite. It measures the performance of the system in integer computation, float point operation, 2D and 3D graphic rendering etc. AnTuTu can provide the overall system impact when the L2 cache is locked.

As shown in Figure. 5.8, locking one out of eight ways in L2 cache has at most 3% performance penalty. However, the overall system performance degrades more quickly when more than 60% of the L2 cache is locked and becomes unavailable. This pattern is consistent with other benchmarks in the AnTuTu suite including 2D GPU, single thread integer operation, and multi-thread integer operation.

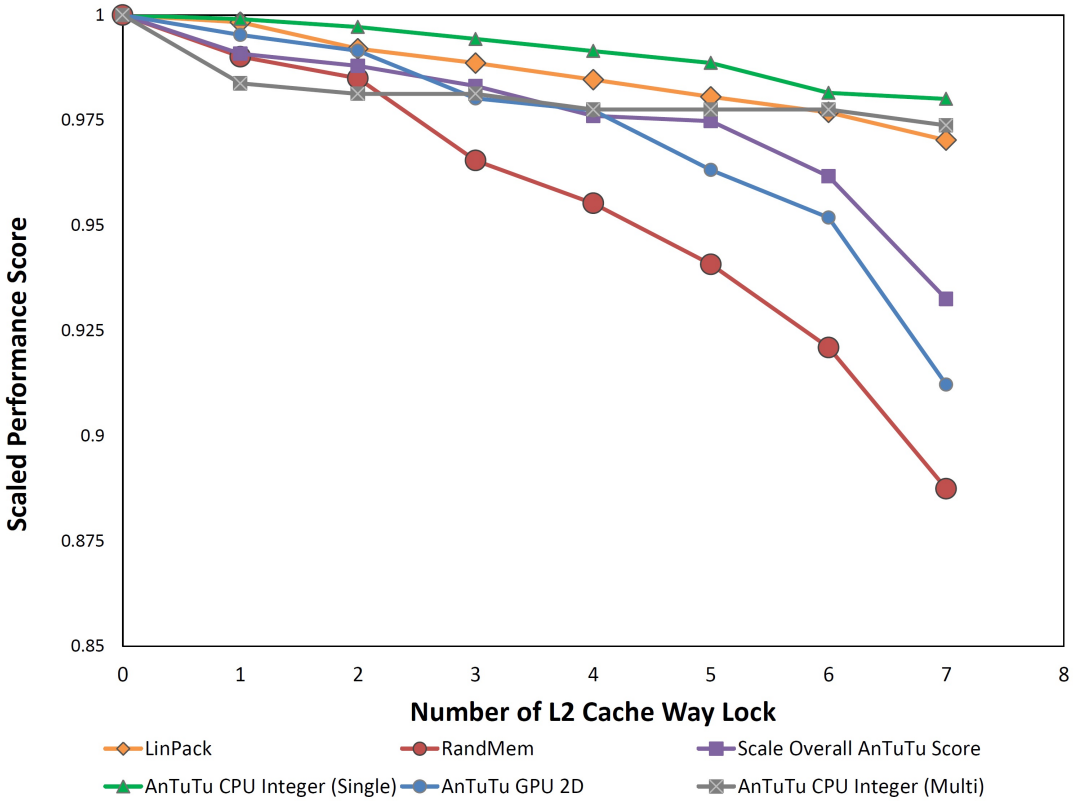

Figure 5.8: Performance Impact of L2 Cache Locking

5.7 Security Analysis

With the physical possession of the mobile device, adversaries have two attack vectors, software attack and cold boot attack. We assume that attackers with physical access can only examine contents of the physical memory but not the cache and registers inside the processor. The attack model is summarized in Table 5.4. The adversary who compromises the rich OS can gain unrestricted read and write access to the normal memory and cache, but neither the secure memory nor the secure cache. The adversary who utilizes cold boot attacks can gain unrestricted read access to the memory of both the normal world and the secure world, but not the cache contents.

CaSE is designed to protect confidentiality and integrity of the application. Applications are encrypted with a checksum while stored in memory. Code and the data of the application are only decrypted in the cache-assisted SoC-bound execution environment.

Attack Vector	S Mem		NS Mem		S Cache		NS Cache	
	Rd	Wr	Rd	Wr	Rd	Wr	Rd	Wr
Software Attack			✓	✓			✓	✓
Cold Boot Attack	✓		✓					

Table 5.4: Attacker Capability on ARM TrustZone

5.7.1 Software Attacks from Compromised Rich OS

In the CaSE execution using normal cache, the rich OS is suspended by disabling all local interrupts. There is no Non-Maskable interrupt (NMI) on the i.MX53. When CaSE is built on platforms with NMI, system designers should redirect these interrupts to TrustZone temporarily for secure processing. With the OS suspended, it cannot launch any attacks to compromise either the integrity or the confidentiality of the processor cache.

In the CaSE execution using secure cache, the rich OS cannot read or modify the secure memory or the secure cache. This is because the memory space is completely separated between the two worlds by TrustZone. To improve performance during context switching, the CaSE execution environment in the secure cache is not sanitized. Because of this design choice, the rich OS may also attempt to use cache maintenance instruction to evict the secure cache out to DRAM, and then use cold boot attack to read out the DRAM contents. However, we verify via experiments that cache maintenance instructions executed in the normal world affect only the normal cache. This is because secure cache is handled differently by the cache controller due to the TrustZone protection.

The rich OS can also launch an impersonation attack. The compromised rich OS can send an application request as the original user. CaSE does not have any built-in mechanism to mitigate this attack. However, an application can use mutual authentication to thaw this attack. For the cryptographic modules in our prototype, adversaries can launch chosen plaintext attack and chosen cipher text attack. However, most modern cryptography methods, including AES, are designed to resist such attacks.

DMA requests on secure memory from peripheral devices, such as LCD controller, are prevented by the TrustZone-aware DMA controller (DMAC). Therefore, the rich OS cannot program peripheral devices in the normal world to read or write secure memory.

Lastly, the compromised rich OS can change the power state without notifying the CaSE controller in the secure world. In this case, all cached sensitive contents of the CaSE controller will be lost. On subsequent invocation to CaSE controller, all requests will be dropped. We do not consider this as a real threat, since the attacker who already has physical access can simply power off the device to deny services.

5.7.2 Unrestricted Memory Read from Cold Boot Attack

Cold boot attacks are capable of reading both the normal world memory and the secure world memory. Since a cold boot attack physically removes DRAM chip from the system, we assume it will be too difficult for the attacker to modify the value in DRAM circuit without interrupting the operation of the system. Therefore, only the confidentiality of the memory is compromised, but not the integrity. In CaSE, application contexts and application binaries are always encrypted while in DRAM. The key for the encryption is stored in the processor cache or in the on-chip secure storage while the system is in power saving mode. Thus, it is protected from cold boot attacks.

Since modern processor cache is built using SRAM which does exhibit the remanence effect similar to DRAM, the compromised OS can attempt to reboot the system to run on a malicious OS to exploit this fact to extract sensitive information from the cache. However, the malicious OS would fail the high assurance booting process. Furthermore, the SoC firmware on the ARM Cortex-A8 processor resets the cache contents upon power reset event. Therefore, we can prevent this attack too.

	Platform		**Software Attack**				**Cold Boot Attack**	
	x86	ARM	Data Confidentiality	Data Integrity	Code Confidentiality	Code Integrity	Data Confidentiality	Code Confidentiality
Hardware-Assisted Execution								
VT-x/AMD-v based [143]	✓		✓	✓		✓		
TXT/AMD SVM based [144]	✓		✓	✓		✓		
SGX based [48]	✓		✓	✓	✓	✓	✓	✓
SMM based [55]	✓		✓	✓		✓		
Coprocessor based [186, 134]	✓		✓	✓		✓	✓	
TrustZone based [177]		✓	✓	✓		✓		
SoC-bound Execution								
Register based [105, 146]	✓	✓					✓	
Cache-based [108, 109, 77]	✓	✓					✓	
OCRAM Based [77]		✓					✓	✓
CaSE		✓	✓	✓	✓	✓	✓	✓

Table 5.5: Comparison of Secure Execution Environment under Software Attack and Cold Boot Attack

5.8 Discussion and Future Work

5.8.1 Migrating CaSE to Other Platforms

Though our prototype implementation of CaSE is on the i.MX53 development board, the system design is widely applicable to other hardware platforms that can provide isolated execution and SoC-bound memory storage.

Multi-core Processors

The i.MX53 has a single core processor. For multi-core processors, it is no longer necessary to suspend the execution of the rich OS. When the SoC-bound execution runs on a subset of the cores, the other cores can continue executing the workloads of the rich OS. It will bring some new challenges. Though applications running in the secure cache can still be protected by the TrustZone isolation, applications running in the normal cache may be compromised by the rich OS running on the other cores. Thus, system designers need to rely on some dedicated system features in the multi-core SoC to enable the cache isolation. For instance, new cache controllers in both ARM and AMD platforms [32, 45] have the capability to assign individual last level cache block to specific processor cores.

On-chip Memory

To improve system performance on embedded devices, SoC designers are continuously increasing the use of silicon layout for allocating more on-chip memory. The average percentage of layout used for memory is 80% in the year 2008 and has been rising [8]. There are three main categories of on-chip memory, SRAM, DRAM, and ROM, where SRAM and DRAM are more suitable to construct SoC-bound execution.

In our implementation of CaSE on the i.MX53, processor cache (SRAM) is used as memory space for SoC-bound execution. There are other choices of on-chip memory as well. For example, the internal DRAM (iRAM) has been used to store cryptographic data of AES [77]. The size of iRAM varies in different systems, and it is usually small. For example, there is only 144 KB iRAM for i.MX53 [33] and 272 KB iRAM for i.MX6 [43]. Furthermore, certain Board Support Package (BSP) uses the iRAM for other purposes such as video processing [14]. Another important difference between iRAM and processor cache is the isolation mechanism. Unlike cache, iRAM occupies a range of physical address space. The protection of this address space from the compromised rich OS or malicious DMA-capable I/O devices might not be available, and it is platform specific [33, 43].

Another important consideration is the remanence effect for on-chip memory storage. On-chip memory is free from the physical memory extraction in cold boot attacks due to its proximity on

die. However, they still exhibit remanence effect just as the external DRAM. When the contents are not sanitized upon system reset, sensitive information could leak out during the boot up process. System designers should verify that the on-chip memory is well protected across different power state changes including full system reset. For example, on the i.MX53, processor firmware resets the cache contents when the processor is rebooted. This can be confirmed by the *L1RSTDISABLE* and *L2RSTDISABLE* bits on the auxiliary control register of the Cortex-A8 processor.

Secure Information Flow

It is critical to secure the information flow between the external DRAM and the on-chip memory when adopting CaSE on other platforms. Our implementation on the i.MX53 relies on the isolation provided by TrustZone and the ability to lock memory contents in cache to prevent malicious attacks. Program contexts are encrypted before being written to the external DRAM. Though cache locking has been supported by a wide range of ARM processors, most x86 processors still do not support fine-grained cache manipulation. On those processors, information flowing out of the processor due to cache contention can be protected by temporal separation, which clears sensitive contents between the two executions of the protected application.

On some new platforms that support I/O coherent cache [78, 34], it is allowed for an I/O device to access cache contents. To protect against potential DMA attacks, system designers can use the inputoutput memory management unit (IOMMU) [35] or the system memory management unit (SMMU) [42] to secure DMA operations.

5.8.2 Supporting Unmodified Applications

CaSE has made it possible to execute arbitrary self-contained applications in a secure execution environment that provides both confidentiality and integrity for the code and data of the application. To support non-trivial unmodified legacy applications, we must enhance the platform with support for encrypted memory paging and verified system calls in the untrusted rich OS.

Encrypted Memory Paging

In our current implementation of CaSE, applications are loaded and decrypted completely within the cache. However, for large size applications that fail to fit in the cache, the current method of SoC-bound execution will not be sufficient. CaSE can be extended to support larger applications by keeping only the most recently used memory pages decrypted inside the SoC while leaving other pages encrypted in memory.

Due to the lack of hardware supported enclave such as Intel SGX [48], memory encryption and decryption will be triggered by the software. In order to provide seamless support for memory paging into and out of the SoC boundary, the page fault handling routine has to be interposed.

When the application accesses a page that is not in the SoC, the page fault can then either be handled by the rich OS [77] or the security monitor in TrustZone [52].

While it is fairly straight forward to extend CaSE to handle applications that do not fit in the cache, similar to other memory encryption system [77, 115, 85] there is a high performance penalty for applications that frequently swap memory pages [77].

Verified System Calls to Rich OS

Many non-trivial applications require OS support to perform meaningful tasks. Our current implementation of CaSE can be further extended to support the use of system call. Before the system call is made, the application is paused and the application context will be encrypted and then stored in DRAM. The system call request is then forwarded to the untrusted rich OS. Upon completion of the system call, the application is resumed by decrypting the application context in the processor cache. Unfortunately, it is not sufficient to simply enable system call from the application. When the OS is compromised, it is possible for the malicious OS to launch Iago attacks [72] where the result of system calls is manipulated to subvert a protected application. Protecting unmodified applications in commodity operating system has been an active area of research [74, 116, 80]. CaSE can benefit from system call behavior verification techniques from these systems [116].

5.9 Related Work

To protect the wide spread of software vulnerabilities in applications and operating systems, hardware-assisted isolation has been widely adopted in both x86 and ARM processors [144, 55, 48, 52, 177]. On the other hand, physical memory disclosure attacks [111, 147] achieve complete memory exposure through a different attack vector. CaSE aims to provide a SoC-bound execution environment that can defend against both attacks. Our work is closely related to the research on isolated execution environments and cold boot resistant computations.

5.9.1 Isolated Execution

A line of research on isolated execution environments [144, 189, 143, 101, 197, 187, 55, 62, 151, 87, 48, 177, 188, 121, 165, 132, 212, 140, 15, 171, 134, 186] has attracted much attention as security becomes one of the most important aspects in modern information systems. One of the key challenges is to bootstrap a trusted environment and to isolate it from the untrusted environment. Earlier work focuses on bootstrapping an isolated environment using a high privileged entity, such as hypervisor [189, 143, 197] or System Management Mode (SMM) [55].

As vulnerabilities are discovered routinely in the highly complex modern OS, hardware-assisted protection is widely used in information systems due to the attractive property of shielding applica-

tions from potentially compromised OS [144, 52, 177, 55, 188, 121, 165]. Intel Trusted Execution Technology (TXT) and AMD Secure Virtual Machine (SVM) are used in [144] to create a trusted isolated execution environment for protecting security sensitive applications. Recognizing the lack of efficient context switching in the TXT technology, Intel recently proposed Intel Secure Guard Extension (SGX) [48] to provide an efficient secure enclave for isolating sensitive applications. Moreover, data stored outside of the processor bound enclave is automatically encrypted by the processor. We share the same design concept as the Intel SGX and target at achieving the same security goals. However, SGX is a processor extension on the x86 platform, while CaSE builds on the commodity TrustZone enabled ARM systems.

For mobile devices that are running on ARM processors, TrustZone has been widely adopted in [177, 188, 121, 165, 132, 212, 140, 15]. Different from the previous works that utilize TrustZone, CaSE attempts to address the threat of cold boot attacks on the system. Lastly, coprocessor has also been proposed to achieve secure computation in adverse environment [171, 134, 186]. CaSE runs on the commodity hardware and does not require additional dedicated coprocessor for code execution.

5.9.2 SoC-bound Execution

In order to defend against cold boot attack, sensitive information has to be kept in memory areas outside the DRAM. There are several types of memory that are inside the physical boundary of the SoC, namely, register, cache, and on-chip RAM. Several research works [105, 146, 170, 99] use register to store cryptographic sensitive materials. In [152, 108, 77, 109], the sensitive cryptographic materials are stored in the processor cache. Alternatively, OCRAM is used in [77]. Sentry [77] is closely related to our work. It also uses cache locking function for CPU-bound execution. However, similar to other cache-based SoC-bound execution, the security of Sentry builds on the strong assumption that the mobile OS can be trusted. We address the risk of compromised OS attack in CaSE. Lastly, even though the use of TrustZone and support for unmodified application are briefly mentioned in [77], no further description of implementations is provided.

Table. 5.5 compares CaSE with other approaches towards secure execution environment in terms of the security protections under software attacks and cold boot attacks. As indicated in the table, CaSE is the first to provide a SoC-bound execution environment on ARM platforms against both cold boot attacks and software attacks. Furthermore, both code and data of the program are protected in CaSE.

5.10 Summary

In this work, we present CaSE, a TrustZone enabled SoC-bound execution environment to protect against both cold boot attack and compromised rich OS attack. CaSE offers two modes of operation, SoC-bound execution in the normal cache and SoC-bound execution in the secure cache,

which provide a trade-off between system performance and security. We build a crypto library and a kernel integrity checker to demonstrate the practical usage of our system on real ARM platform. The experimental results show that CaSE environment only introduces little performance overhead.

Chapter 6

Conclusion

The ability to access any information at the finger tips anywhere in a matter of seconds has redefined the mode of operation for the daily lives. Yet this new paradigm of rapid information flow also brings serious concern of security and privacy risk uniquely to this age of information. With the ever increasing persistent cyber attacks that organizations and agencies face nowadays, it is more important than ever to be able to operate in a secure manner in adversarial environment.

6.1 Research Summary

In this dissertation, I have explored security in adversarial environment spanning a broad spectrum from the cloud computing to desktop and mobile end points. Advance system attack and defense are proposed to further our understanding in the principles of system security and to shape the future of computer security. Specifically, we have the following findings.

- In TUDEC [207, 206], we study a trusted execution environment optimized to provide close-to-bare metal performance for data-intensive execution in the cloud. Instead of performing computation on encrypted data, which has been recognized as prohibitively expensive, we propose to perform computation on decrypted user data inside a trusted execution environment. This isolated environment is capable of safeguarding user data even when the software on the neighboring virtual machines or host is compromised. We've built a prototype and have demonstrated that the TUDEC has significant improvements over device emulation in the previous trusted isolated environment, thus more suitable for data-intensive application.
- In HIveS [209], we propose a different approach to anti-memory forensic. Instead of looking at ways to conceal presence by operating system object manipulation, we can defeat current memory acquisition methods by manipulating the physical address layout, a design architectural feature on modern x86 platforms. HIveS is an anti-forensic mechanism to conceal in-memory data shadowed

behind the I/O address space. Besides I/O Shadowing technique to prevent forensic memory acquisition tools from reading the HIveS memory contents via processor, we also use Blackbox Write and TLB Camouflage to enable the attacker exclusive write access and provide a single malicious core exclusive read and write access, respectively. Furthermore, we propose several add-ons to the basic framework to further hide from physical memory forensics. A prototype of HIveS is built on an AMD platform to show that none of the popular memory acquisition tools we tested can capture the memory data protected by HIveS. Several countermeasures are discussed in the end.

- In CacheKit [208], 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.

- In CaSE [210], we study a TrustZone enabled SoC-bound execution environment to protect against both cold boot attack and compromised rich OS attack. CaSE offers two modes of operation, SoC-bound execution in the normal cache and SoC-bound execution in the secure cache, which provide a trade-off between system performance and security. We build a crypto library and a kernel integrity checker to demonstrate the practical usage of our system on real ARM platform. The experimental results show that CaSE environment only introduces little performance overhead.

6.2 Future Work

Many topics explored in this dissertation can be further extended as following.

- TUDEC makes use of the system management mode in the BIOS to provide an isolated execution environment, and the TPM to provide the root of trust. It would be interesting to apply TUDEC to other platforms where it can be integrated as a UEFI module. Furthermore, different types of hardware supported isolated execution models can be applied in the TUDEC replacing system management mode.

- HIveS provides anti-forensic storage within the x86 system. In particular, the use of IORR register is explored in the dissertation. Even though IORR is used effectively to redirect memory access in the HIveS rootkit, the use of the MSRs can become a signal for system defenders. It would be interesting to explore the possibility of using other hardware configurations besides the registers on the processor to redirect memory reads and writes.

- CacheKit explores the fundamental issues of privilege separated cache in introspection. However, even though cache is used to store sensitive malicious code in the prototype, the use of hardware supported cache locking mechanism still hinders wide adoption of the method on different systems. It would be interesting to further explore other methods to maintain code in cache, besides hardware supports.
- CaSE presents a cache based secure execution framework supported by TrustZone. However, the current design places the entire sensitive application inside the cache, and furthermore does not support any system calls. It would be interesting to explore the possibility of encrypting the memory to accommodate larger applications and methods to utilize the untrusted rich OS in CaSE.

Bibliography

[1] 3rd generation intel xscale microarchitecture developers 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.

[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] Core boot. http://www.coreboot.org.

[8] Everything you wanted to know about soc memory. http://www.low-powerdesign.com/Everything_You_Wanted_to_Know_About_SOC_Memory.pdf.

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

[10] Kvm. http://www.linux-kvm.org/page/Main_Page.

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

[12] National vulnerability database. http://nvd.nist.gov/.

[13] Pci sig: Pci-sig single root i/o virtualization. http://www.pcisig.com/specifications/iov/single_root.

[14] Reference bsps for freescale i.mx53 quick start board. http://www.adeneo-embedded.com/Products/Board-Support-Packages/Freescale-i.MX53-QSB. Accessed: 2015-04-30.

[15] Samsung knox. https://www.samsungknox.com/en.

[16] secondlook. http://secondlookforensics.com/.

[17] Sierraware. http://www.sierraware.com/open-source-ARM-TrustZone.html.

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

[19] Trusted computing group. http://www.trustedcomputinggroup.org/.

[20] Uefi specification. http://www.uefi.org/home.

[21] Vmware. http://www.vmware.com.

[22] volatility. https://code.google.com/p/volatility/.

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

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

[25] Intel Chipset 4 GB System Memory Support . Feb 2005.

[26] Amd-v nested paging, rev 1.0. july 2008.

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

[28] Iommu architectural specification, rev 1.26. February 2009.

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

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

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

[32] *CoreLink Level 2 Cache Controller L2C-310, Technical Reference Manual*, June 2012.

[33] *i.MX53 Multimedia Applications Processor Reference Manual*, June 2012.

[34] *Intel Data Direct I/O Technology (Intel DDIO) A Primer*, Feb 2012.

[35] Intel 64 and ia-32 architectures software developers manual. August 2012.

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

[37] Vulnerability report for xen 4.x. http://secunia.com/advisories/product/33176/?task=advisories_2012, February 2012.

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

[39] *ARM Cortex-A15 MPCore Processor, Technical Reference Manual*, june 2013.

[40] Arm strategic report. 2014.

[41] 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`, 2014.

[42] *ARM System Memory Management Unit Architecture Specification - SMMU architecture version 2.0*, July 2015.

[43] *i.MX 6Dual/6Quad Applications Processor Reference Manual*, July 2015.

[44] Mobile threat report: Whats on the horizon for 2016. 2015.

[45] Advanced Micro Devices, Inc. BIOS and Kernel Developer's Guide (BKDG) For AMD Family 15h Processors, Rev 3.23.

[46] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security. *ARM white paper*, 3(4), 2004.

[47] Kapil Anand and Rajeev 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.

[48] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, page 10, 2013.

[49] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[50] Mikhail J. Atallah and Jiangtao Li. Secure outsourcing of sequence comparisons. *Int. J. Inf. Secur.*, 4(4):277–287, October 2005.

[51] Ahmed M Azab, Peng Ning, Emre Can Sezer, and Xiaolan Zhang. Hima: A hypervisor-based integrity measurement agent. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 461–470. IEEE, 2009.

[52] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo 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.

[53] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. CCS '10, pages 38–49, New York, NY, USA, 2010. ACM.

[54] Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. CCS '11, pages 375–388, New York, NY, USA, 2011. ACM.

[55] Ahmed M Azab, Peng Ning, and Xiaolan Zhang. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 375–388. ACM, 2011.

[56] Ashar Aziz. The evolution of cyber attacks and next generation threat protection. *RSA Conference*, 2013.

[57] Paul Bakker. Polarssl. `https://github.com/ARMmbed/mbedtls`. Accessed: 2015-04-30.

[58] Edward Balas. sebek, 2005.

[59] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 77–86. IEEE, 2008.

[60] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[61] Andrew Baumann, Marcus Peinado, and Galen 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.

[62] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 267–283. USENIX Association, 2014.

[63] Raytheon BBN. Fred: Forensic ram extraction device. `http://www.digitalintelligence.com/products/fred/`.

[64] Michael Becher, Maximillian Dornseif, and Christian N Klein. FireWire All Your Memory are Belong to us. *Proceedings of CanSecWest*, 2005.

[65] Nicole Beebe. Digital forensic research: The good, the bad and the unaddressed. In *Advances in digital forensics V*, pages 17–36. Springer, 2009.

[66] Darren Bilby. Low down and dirty: Anti-forensic rootkits. *BlackHat Japan*, 2006.

[67] Erik-Oliver Blass and William Robertson. Tresor-hunt: attacking cpu-bound encryption. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 71–78. ACM, 2012.

[68] Daniel Bovet and Marco Cesati. *Understanding the Linux kernel*. O'reilly, 2007.

[69] Brian D. Carrier and Joe Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50 – 60, 2004.

[70] Ellick Chan, Shivaram Venkataraman, Francis David, Amey Chaugule, and Roy Campbell. Forenscope: A framework for live forensics. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 307–316. ACM, 2010.

[71] Ellick M. Chan, Jeffrey C. Carlyle, Francis M. David, Reza Farivar, and Roy H. Campbell. Bootjacker: Compromising computers using forced restarts. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, 2008.

[72] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. *SIGARCH Comput. Archit. News*, 41(1):253–264, March 2013.

[73] Xiaofeng Chen, Jin Li, Jianfeng Ma, Qiang Tang, and Wenjing Lou. New algorithms for secure outsourcing of modular exponentiations. In *ESORICS*, pages 541–556, 2012.

[74] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 2–13. ACM, 2008.

[75] Richard Chow, Philippe Golle, Markus Jakobsson, Elaine Shi, Jessica Staddon, Ryusuke Masuoka, and Jesus Molina. Controlling data in the cloud: outsourcing computation without outsourcing control. CCSW '09, pages 85–90, New York, NY, USA, 2009. ACM.

[76] MI Cohen, D Bilby, and G Caronni. Distributed forensics and incident response in the enterprise. *digital investigation*, 8:S101–S110, 2011.

[77] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal de Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Protecting data on smartphones and tablets from memory attacks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 177–189. ACM, 2015.

[78] ARM Cortex. A9 technical reference manual, 2012.

[79] National Research Council. Strengthening Forensic Science in the United States: A Path Forward. https://www.ncjrs.gov/pdffiles1/nij/grants/228091.pdf, 2009.

[80] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual ghost: Protecting applications from hostile operating systems. *ACM SIGARCH Computer Architecture News*, 42(1):81–96, 2014.

[81] Ang Cui, Michael Costello, and Salvatore J Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *NDSS*, 2013.

[82] Francis M David, Ellick M Chan, Jeffrey C Carlyle, and Roy H Campbell. Cloaker: Hardware supported rootkit concealment. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 296–310. IEEE, 2008.

[83] Bryan Dixon and Shivakant 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.

[84] Jack Dongarra and Piotr Luszczek. Linpack benchmark. *Encyclopedia of Parallel Computing*, pages 1033–1036, 2011.

[85] Guillaume Duc and Ronan Keryell. Cryptopage: an efficient secure architecture with memory encryption, integrity and information leakage protection. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 483–492. IEEE, 2006.

[86] Shawn Embleton, Sherri Sparks, and Cliff C Zou. Smm rootkit: a new breed of os independent malware. *Security and Communication Networks*, 6(12):1590–1605, 2013.

[87] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. Iso-x: A flexible architecture for hardware-managed isolated execution. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 190–202. IEEE, 2014.

[88] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 5, 2011.

[89] Dan Farmer and Wietse Venema. *Forensic discovery*, volume 18. Addison-Wesley Reading, 2005.

[90] Peter Ferrie. Attacks on more virtual machine emulators.

[91] Elia Florio. When malware meets rootkits. *Virus Bulletin*, 2005.

[92] Marc Fossi, Gerry Egan, Kevin Haley, Eric Johnson, Trevor Mack, Téo Adams, Joseph Blackbird, Mo King Low, Debbie Mazurek, David McKinney, et al. Symantec internet security threat report trends for 2010. *Volume*, 16:20, 2011.

[93] Susannah Fox. 51% of u.s. adults bank online. http://www.pewinternet.org/2013/08/07/51-of-u-s-adults-bank-online/.

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

[95] Simson L Garfinkel. Digital forensics research: The next 10 years. *Digital Investigation*, 7:S64–S73, 2010.

[96] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is not transparency: Vmm detection myths and realities. In *HotOS*, 2007.

[97] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. *SIGOPS Oper. Syst. Rev.*, 37(5):193–206, October 2003.

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

[99] Behrad Garmany and Tilo Müller. Prime: private rsa infrastructure for memory-less encryption. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 149–158. ACM, 2013.

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

[101] Peter Gilbert, Landon P Cox, Jaeyeon Jung, and David Wetherall. Toward trustworthy mobile sensing. In *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*, pages 31–36. ACM, 2010.

[102] Guy Gogniat, Tilman Wolf, Wayne Burleson, Jean-Philippe Diguet, Lilian Bossuet, and Romain Vaslin. Reconfigurable hardware for high-security/high-performance embedded systems: the safes perspective. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(2):144–155, 2008.

[103] Shyamnath Gollakota, Haitham Hassanieh, Benjamin Ransford, Dina Katabi, and Kevin Fu. They can hear your heartbeats: non-invasive security for implantable medical devices. *ACM SIGCOMM Computer Communication Review*, 41(4):2–13, 2011.

[104] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. Eli: bare-metal performance for i/o virtualization. ASPLOS XVII, pages 411–422, New York, NY, USA, 2012. ACM.

[105] Johannes Gotzfried and Tilo Muller. Armored: Cpu-bound encryption for android-driven arm devices. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 161–168. IEEE, 2013.

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

[107] Andrew Griffiths. Binary protection schemes. `https://www.exploit-db.com/docs/59.pdf`. Accessed: 2016-03-01.

[108] Le Guan, Jingqiang Lin amd Bo Luo, and Jiwu Jing. Copker: Computing with Private Keys without RAM. In *In Network and Distributed System Security Symposium (NDSS)*, 2014.

[109] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 3–19, May 2015.

[110] Anthony Gutierrez, Ronald G Dreslinski, Thomas F Wenisch, Trevor Mudge, Ali Saidi, Chris Emmons, and Nigel Paver. Full-system analysis and characterization of interactive smartphone applications. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 81–90. IEEE, 2011.

[111] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[112] Ryan Harris. Arriving at an anti-forensics consensus: Examining how to define and control the anti-forensics problem. *digital investigation*, 3:44–49, 2006.

[113] Takahiro Haruyama and Hiroshi Suzuki. One-byte modifications for breaking memory forensic analysis. *Black Hat Europe*, 2012.

[114] John Heasman. Implementing and detecting a pci rootkit. 20(2007):3, 2006.

[115] Michael Henson and Stephen Taylor. Memory encryption: a survey of existing techniques. *ACM Computing Surveys (CSUR)*, 46(4):53, 2014.

[116] Owen S Hofmann, Sangman Kim, Alan M Dunn, Michael Z Lee, and Emmett Witchel. Inktag: Secure applications on an untrusted operating system. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 265–278. ACM, 2013.

[117] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 191–205. IEEE, 2013.

[118] MessageLabs Intelligence. Annual security report. *Symantec Corp*, 2014.

[119] G. Irazoqui, T. Eisenbarth, and B. Sunar. S$a: A shared cache attack that works across cores and defies vm sandboxing – and its application to aes. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 591–604, May 2015.

[120] Daehee Jang, Hojoon Lee, Minsu Kim, Daehyeok Kim, Daegyeong Kim, and Brent Byunghoon Kang. Atra: Address translation redirection attack against hardware-based external monitors. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 167–178. ACM, 2014.

[121] Jinsoo Jang, Sunjune Kong, Minsu Kim, Daegyeong Kim, and Brent Byunghoon Kang. Secret: Secure channel between rich execution environment and trusted execution environment. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS'15, 2015.

[122] Xuxian Jiang, Xinyuan Wang, and Dongyan 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.

[123] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. Nohype: virtualized cloud infrastructure without the virtualization. ISCA '10, pages 350–361, New York, NY, USA, 2010. ACM.

[124] Gene H Kim and Eugene 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.

[125] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. System-level protection against cache-based side channel attacks in the cloud. In *USENIX Security*, 2012.

[126] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

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

[128] Kostya Kortchinsky. Cloudburst: A vmware guest to host escape. *Black Hat Conference*, 2009.

[129] Éric Lacombe, Frédéric Raynal, and Vincent Nicomette. Rootkit modeling and experiments under linux. *Journal in Computer Virology*, 4(2):137–157, 2008.

[130] Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *Security & Privacy, IEEE*, 9(3):49–51, 2011.

[131] Ming Li, Shucheng Yu, Ning Cao, and Wenjing Lou. Authorized private keyword search over encrypted data in cloud computing. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, 2011.

[132] Wenhao Li, Haibo Li, Haibo Chen, and Yubin 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.

[133] Eugene Libster and Jesse D. Kornblum. A proposal for an integrated memory acquisition mechanism. *SIGOPS Oper. Syst. Rev.*, 42(3):14–20, April 2008.

[134] M Lindemann, Roxana Perez, Rudolf Sailer, L Van Doorn, and Sean W Smith. Building the ibm 4758 secure coprocessor. *Computer*, 34(10):57–66, 2001.

[135] Jiuxing Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 gbe nics with sr-iov support. pages 1–12, 2010.

[136] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. High performance vmm-bypass i/o in virtual machines. ATEC '06, pages 3–3, Berkeley, CA, USA, 2006. USENIX Association.

[137] Roy Longbottom. Roy longbottoms pc benchmark collection, 2014.

[138] Yinghai Lu, Li-Ta Lo, Gregory R. Watson, and Ronald G. Minnich. CAR: Using Cache as RAM in LinuxBIOS. `http://rere.qmqm.pl/~mirq/cache_as_ram_lb_09142006.pdf`, 2006.

[139] Yinghai Lu, LiTa Lo, Gregory Watson, and Ronald Minnich. CAR: Using Cache as RAM in LinuxBIOS. `http://rere.qmqm.pl/~mirq/cache_as_ram_lb_09142006.pdf`.

[140] Claudio Marforio, Nikolaos Karapanos, Claudio Soriente, Kari Kostiainen, and Srdjan Capkun. Smartphones as practical and secure location verication tokens for payments. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS'14, 2014.

[141] Lorenzo Martignoni, Aristide Fattori, Roberto Paleari, and Lorenzo Cavallaro. Live and trustworthy forensic analysis of commodity production systems. In *Recent Advances in Intrusion Detection*, pages 297–316. Springer, 2010.

[142] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. SP '10, pages 143–158, Washington, DC, USA, 2010. IEEE Computer Society.

[143] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 143–158. IEEE, 2010.

[144] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for tcb minimization. *SIGOPS Oper. Syst. Rev.*, 42(4):315–328, April 2008.

[145] Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. Vigilare: Toward snoop-based kernel integrity monitor. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[146] Tilo Müller, Felix C Freiling, and Andreas Dewald. Tresor runs encryption securely outside ram. In *USENIX Security Symposium*, 2011.

[147] Tilo Müller and Michael Spreitzenbarth. Frost. In *Applied Cryptography and Network Security*, pages 373–388. Springer, 2013.

[148] Tim Newsham, Chris Palmer, Alex Stamos, and Jesse Burns. Breaking forensics software: Weaknesses in critical evidence collection. In *Proceedings of the 2007 Black Hat Conference*, 2007.

[149] Jon Oberheide, Kaushik Veeraraghavan, Evan Cooke, Jason Flinn, and Farnam 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.

[150] MFXJ Oberhumer, László Molnár, and John F Reiser. Upx: Ultimate packer for executables, 2004.

[151] Emmanuel Owusu, Jorge Guajardo, Jonathan McCune, Jim Newsome, Adrian Perrig, and Amit Vasudevan. Oasis: On achieving a sanctuary for integrity and secrecy on untrusted platforms. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 13–24. ACM, 2013.

[152] J. Pabel. Frozencache: Mitigating cold-boot attacks for full-disk-encryption software. In *27th Chaos Communication Congress*, 2010.

[153] Bryan D Payne, Martim Carbone, Monirul Sharif, and Wenke 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.

[154] Nick L Petroni Jr, Timothy Fraser, Jesus Molina, and William A Arbaugh. Copilot-a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, pages 179–194, 2004.

[155] Ian Pratt, Keir Fraser, Steven Hand, Christian Limpach, Andrew Warfield, Dan Magenheimer, Jun Nakajima, and Asit Mallick. Xen 3.0 and the Art of Virtualization. In *Proceedings of the 2005 Ottawa Linux Symposium*, July 2005.

[156] Nguyen Anh Quynh and Yoshiyasu Takefuji. Towards a tamper-resistant kernel rootkit detector. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 276–283. ACM, 2007.

[157] J. ruthkowska R Wojtczuk. Xen 0wning triology. *Black Hat Conference*, 2008.

[158] Alessandro Reina, Aristide Fattori, Fabio Pagani, Lorenzo Cavallaro, and Danilo Bruschi. When hardware meets software: A bulletproof solution to forensic memory acquisition. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 79–88, New York, NY, USA, 2012. ACM.

[159] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[160] J. M. Rushby. Design and verification of secure systems. SOSP '81, pages 12–21, New York, NY, USA, 1981. ACM.

[161] J. Rutkowska. Beyond the CPU: Defeating hardware based RAM acquisition. *Proceedings of BlackHat DC 2007*, 2007.

[162] Joanna Rutkowska. Subverting vistatm kernel for fun and profit. *Black Hat Briefings*, 2006.

[163] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert Van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *USENIX Security Symposium*, volume 13, pages 223–238, 2004.

[164] F.L. Sang, E. Lacombe, V. Nicomette, and Y. Deswarte. Exploiting an i/ommu vulnerability. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 7–14, 2010.

[165] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec 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.

[166] Bradley Schatz. Bodysnatcher: Towards reliable volatile memory acquisition by software. *digital investigation*, 4:126–134, 2007.

[167] sd. Linux on-the-fly kernel pactching. phrack, 2002.

[168] Devik Sd. Linux on-the-fly kernel patching without lkm. *Volume 0x0b, Issue 0x3a, Phile# 0x07 of 0x0e-Phrack Magazine-http://www. phrack-dont-give-a-shit-about-dmca. org/show. php*, 2001.

[169] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. SOSP '07, pages 335–350, New York, NY, USA, 2007. ACM.

[170] Patrick Simmons. Security through amnesia: a software-based solution to the cold boot attack on disk encryption. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 73–82. ACM, 2011.

[171] Sean W Smith and Steve Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8):831–860, 1999.

[172] Sherri Sparks and Jamie Butler. Shadow walker: Raising the bar for rootkit detection. *Black Hat Japan*, pages 504–533, 2005.

[173] Udo Steinberg and Bernhard Kauer. Nova: a microhypervisor-based secure virtualization architecture. EuroSys '10, pages 209–222, New York, NY, USA, 2010. ACM.

[174] Patrick Stewin and Iurii Bystrov. Understanding dma malware. DIMVA'12, pages 21–41, Berlin, Heidelberg, 2013. Springer-Verlag.

[175] Johannes Stüttgen and Michael Cohen. Anti-forensic resilient memory acquisition. *Digital Investigation*, 10:S105–S115, 2013.

[176] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Sushil Jajodia. Trustdump: Reliable memory acquisition on smartphones. In *Computer Security-ESORICS 2014*, pages 202–218. Springer, 2014.

[177] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*, pages 367–378. IEEE, 2015.

[178] Kun Sun, Jiang Wang, Fengwei Zhang, and Angelos Stavrou. Secureswitch: Bios-assisted isolation and switch between trusted and untrusted commodity oses. In *NDSS*, 2012.

[179] Wenhai Sun, Bing Wang, Ning Cao, Ming Li, Wenjing Lou, Y. Thomas Hou, and Hui Li. Privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking. ASIA CCS '13, pages 71–82, New York, NY, USA, 2013. ACM.

[180] Joe Sylve. Lime-linux memory extractor. *ShmooCon12*, 2012.

[181] Jakub Szefer, Eric Keller, Ruby B. Lee, and Jennifer Rexford. Eliminating the hypervisor attack surface for a more secure cloud. CCS '11, pages 401–412, New York, NY, USA, 2011. ACM.

[182] Christ Tarnovsky. Attacking hardware: Unsecuring [once] secure devices. 2009.

[183] Sun Tzu. *The art of war*. Orange Publishing, 2013.

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

[185] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[186] Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Pixelvault: Using gpus for securing cryptographic operations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1131–1142. ACM, 2014.

[187] Amit Vasudevan, Jonathan McCune, James Newsome, Adrian Perrig, and Leendert 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.

[188] Amit Vasudevan, Emmanuel Owusu, Zongwei Zhou, James Newsome, and Jonathan M. McCune. Trustworthy execution on mobile devices: What security properties can my mobile platform give me? In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing (Trust 2012)*, June 2012.

[189] Amit Vasudevan, Bryan Parno, Ning Qu, Virgil D. Gligor, and Adrian Perrig. Lockdown: Towards a safe and practical architecture for security applications on commodity platforms. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12, pages 34–54, Berlin, Heidelberg, 2012. Springer-Verlag.

[190] Stefan Vömel and Felix C Freiling. A survey of main memory acquisition and analysis techniques for the windows operating system. *Digital Investigation*, 8(1):3–22, 2011.

[191] Cong Wang, Kui Ren, and Jia Wang. Secure and practical outsourcing of linear programming in cloud computing. In *INFOCOM, 2011 Proceedings IEEE*, pages 820–828, 2011.

[192] Cong Wang, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-preserving public auditing for data storage security in cloud computing. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9, 2010.

[193] Jiang Wang, Angelos Stavrou, and Anup K. Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *RAID*, pages 158–177, 2010.

[194] Jiang Wang, Fengwei Zhang, Kun Sun, and Angelos Stavrou. Firmware-assisted memory acquisition and analysis tools for digital forensics. In *Systematic Approaches to Digital Forensic Engineering (SADFE), 2011 IEEE Sixth International Workshop on*, pages 1–5. IEEE, 2011.

[195] Yi-Min Wang, Doug Beck, Binh Vo, Roussi Roussev, and Chad Verbowski. Detecting stealth software with strider ghostbuster. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 368–377. IEEE, 2005.

[196] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. SP '10, pages 380–395, Washington, DC, USA, 2010. IEEE Computer Society.

[197] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 380–395. IEEE, 2010.

[198] Zhi Wang, Xuxian Jiang, Weidong Cui, and Xinyuan Wang. Countering persistent kernel rootkits through systematic hook discovery. In *Recent Advances in Intrusion Detection*, pages 21–38. Springer, 2008.

[199] Lei Wei and Michael K. Reiter. Third-party private dfa evaluation on encrypted files in the cloud. In *ESORICS*, pages 523–540, 2012.

[200] Paul Willmann, Scott Rixner, and Alan L. Cox. Protection strategies for direct access to virtualized i/o devices. ATC'08, pages 15–28, Berkeley, CA, USA, 2008. USENIX Association.

[201] Johannes 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.

[202] Rafal Wojtczuk and Joanna Rutkowska. Attacking SMM Memory via Intel CPU Cache Poisoning, March 2009.

[203] Zhen Xu, Cong Wang, Qian Wang, Kui Ren, and Lingyu Wang. Proof-carrying cloud computation: The case of convex optimization. In *INFOCOM, 2013 Proceedings IEEE*, pages 610–614, 2013.

[204] Miao Yu, Qian Lin, Bingyu Li, Zhengwei Qi, and Haibing Guan. Vis: virtualization enhanced live acquisition for native system. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 13. ACM, 2011.

[205] Shucheng Yu, Cong Wang, Kui Ren, and Wenjing Lou. Achieving secure, scalable, and fine-grained data access control in cloud computing. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9, 2010.

[206] Ning Zhang, Ming Li, Wenjing Lou, and Y Thomas Hou. Mushi: Toward multiple level security cloud with strong hardware level isolation. In *Military Communications Conference, 2012-MILCOM 2012*, pages 1–6. IEEE, 2012.

[207] Ning Zhang, Wenjing Lou, Xuxian Jiang, and Y Thomas Hou. Enabling trusted data-intensive execution in cloud computing. In *Communications and Network Security (CNS), 2014 IEEE Conference on*, pages 355–363. IEEE, 2014.

[208] Ning Zhang, He Sun, Kun Sun, Wenjing Lou, and Y Thomas Hou. Cachekit: Evading memory introspection using cache incoherence. In *Security and Privacy (SP), 2016 IEEE European Symposium*. IEEE, 2016.

[209] Ning Zhang, Kun Sun, Wenjing Lou, Tom Hou, and Jajodia 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.

[210] Ning Zhang, Kun Sun, Wenjing Lou, and Y Thomas Hou. Case: Cache-assisted secure execution. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016.

[211] Xiaolan Zhang, Leendert van Doorn, Trent Jaeger, Ronald Perez, and Reiner Sailer. Secure coprocessor-based intrusion detection. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 239–242. ACM, 2002.

[212] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi 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.

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