Hardware Mechanisms for Efficient Memory System Security

by

Salessawi Ferede Yitbarek

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2018

Doctoral Committee:
Professor Todd M. Austin, Chair
Assistant Professor Reetuparna Das
Professor Scott Mahlke
Professor Dennis M. Sylvester

Salessawi Ferede Yitbarek

salessaf@umich.edu

ORCID iD: 0000-0003-1420-7937

In loving memory of Haileleul Mulugeta Haile.

Acknowledgements

This thesis would not have been possible without the support of numerous people.

I have been extremely fortunate to have Professor Todd Austin as an advisor. Todd has taught me a great deal about doing research, crafting effective presentations, leading projects, and more. He has been an incredibly supportive advisor who consistently took time out of his busy schedule to mentor me on subjects beyond our research.

I would also like to thank my thesis committee members, Professors Reetuparna Das, Scott Mahlke, and Dennis Sylvester for their valuable feedback. I had the pleasure of collaborating with Reetuparna on multiple projects as well, and her rigorous approach to research has deeply shaped me. Professor Valeria Bertacco has also been an excellent source of wisdom on numerous subjects ranging from communicating my research to surviving graduate school and navigating the academic world.

I have benefited a great deal from all the graduate students I have worked and interacted with. Misiker Aga and David Williams did a lot of heavy-lifting for the infrastructure used for the DDR3 scrambler analysis in Chapter 3. Doowon Lee has kindly shared his expertise with me while I was setting up the experiments and writing this document. My work has also greatly benefited from discussions and feedbacks from Shazeen Aga. Thank you to Lauren Biernacki, Zelalem Birhanu, William Ehrett, Mark Gallagher, Vidushi Goyal, Colton Holoday, Abraham Lamesgin, Timothy Linscott, Andrew McCrabb, Hiwot Tadese, Tarunesh Verma, and Brendan West, for all their valuable feedbacks, and for being a constant source of fun around the lab. I have been fortunate to collaborate on various projects outside of this dissertation and learn from an outstanding set of students, including Zelalem Birhanu, Mark Gallagher, Lauren Biernacki, Tao Yang, Evan Chavis, Kegan Thorrez, Kartik Joshi, and Walter Zarate. Thank you to Rawan Abdel Khalek and Ritesh Parikh for giving me valuable direction as I was starting out. Thank you to William Arthur, who guided me as I was settling and finding my way in the A-Lab. Thank you to Patipan Prasertsom, Ram Srivatsa,

Thomas Zachariah, and Animesh Jain for the loads of fun memories and labor we shared. Thank you to Sai Gouravajhala for being a fantastic companion during all the stressful semesters. I would especially like to thank my friend Biruk Wendimagegn, who has supported me in numerous ways since day one, from helping me settle in Ann Arbor to helping me gain my footing as a Ph.D. student to consulting me as I try to make post-graduation plans. It would be impossible to list all the students I have been enriched by during my stay at the University of Michigan.

I would also like to thank administrative staff members Ashley Andreae, Christine Boltz, Alice Melloni, Alexis Santa Cruz, Karen Liska, Denise Duprie, Stephen Reger, Jamie Goldsmith, Dawn Freysinger, Lauri Johnson-Rafalski, and Laura Pasek for tirelessly assisting me with all the administrative processes I needed to go through.

Thank you to Kidus Ayalneh, Misiker Tadesse, Zelalem Birhanu, Abraham Lamesgin, Helen Arefayne, Hiwot Tadese, Abeselom Fanta, Zerihun Bekele, Abdi Zeynu, Yodit Beyene, and Leul Beyene for the fun, friendship, and support. I have also been immensely blessed by my friendships with Daniel McCarter, Jazmine and Stephen Xu, Ciara Reyes, Carole and Derek Metzger, Carmen Lee, Joel Perry, and Gee Zhu. Thank you to Meseret Tadesse, the Mekonene's (Michael, Tsedey, Hanna, Selamawit, and Mekonnen), Meshesha Mengesha, Genet Girma, Girma Moges, and Bethlehem Yikono for all they have done to make me feel at home. I am also grateful for the many years of continued companionship and emotional support from my closest friends back home: Abisa Girma, Sara Abdella, Mistere Mamo, Lensa Teklu, Sara Teklu, and Besufekad Mamo.

I am indebted to my parents, Adamwork and Ferede, who have always gone out of their way to support me and enable me to follow my interests. Thank you to Zemen Ferede, Ewnetu Ferede, and Meron Mekonen for always looking out for me and for their unwavering support. I am grateful my brother Ewnetu introduced me to programming and encouraged me to pursue my curiosities. Thank you to Haileluel Mulugeta who always encouraged me to pursue a graduate degree and supported me as I was moving to the U.S. to work on my Ph.D. I have looked up to him ever since I got to know him, and I will dearly miss him. Thank you to Abeba Beyene and Dejene Shewaye for their emotional support. I cannot thank my beloved Bethlehem enough for being an unending source of optimism, encouragement, and support, and for making this journey delightful.

Table of Contents

List of Figures

Figure

List of Tables

Table

Abstract

The security of a computer system hinges on the trustworthiness of the operating system and the hardware, as applications rely on them to protect code and data. As a result, multiple protections for safeguarding the hardware and OS from attacks are being continuously proposed and deployed. These defenses, however, are far from ideal as they only provide partial protection, require complex hardware and software stacks, or incur high overheads. This dissertation presents hardware mechanisms for *efficiently* providing *strong* protections against an array of attacks on the memory hardware and the operating system's code and data.

In the first part of this dissertation, we analyze and optimize protections targeted at defending memory hardware from physical attacks. We begin by showing that, contrary to popular belief, current DDR3 and DDR4 memory systems that employ memory scrambling are still susceptible to cold boot attacks (where the DRAM is frozen to give it sufficient retention time and is then re-read by an attacker after reboot to extract sensitive data). We then describe how memory scramblers in modern memory controllers can be transparently replaced by strong stream ciphers without impacting performance.

We also demonstrate how the large storage overheads associated with authenticated memory encryption schemes (which enable tamper-proof storage in off-chip memories) can be reduced by leveraging compact integer encodings and error-correcting code (ECC) DRAMs – without forgoing the error detection and correction capabilities of ECC DRAMs.

The second part of this dissertation presents Neverland: a low-overhead, hardware-assisted, memory protection scheme that safeguards the operating system from rootkits

and kernel-mode malware. Once the system is done booting, Neverland's hardware takes away the operating system's ability to overwrite certain configuration registers, as well as portions of its own physical address space that contain kernel code and security-critical data. Furthermore, it prohibits the CPU from fetching privileged code from any memory region lying outside the physical addresses assigned to the OS kernel and drivers. This combination of protections makes it extremely hard for an attacker to tamper with the kernel or introduce new privileged code into the system – even in the presence of software vulnerabilities. Neverland enables operating systems to reduce their attack surface without having to rely on complex integrity monitoring software or hardware.

The hardware mechanisms we present in this dissertation provide building blocks for constructing a secure computing base while incurring lower overheads than existing protections.

Chapter 1

Introduction

The security of a computer system hinges on the trustworthiness of the operating system and the hardware, as applications rely on them to protect code and data. As a result, modern systems employ multiple layers of hardware-level and software-level defenses to ensure the integrity of the hardware and the kernel. Today's CPUs, for example, provide protections against hardware-level bus snooping and memory tampering attacks, can prevent the execution of arbitrary code in kernel-mode, and allow execution of software components in a hardware-isolated environment. Operating systems are also being shipped with increasingly sophisticated attack mitigations. For example, it is becoming a common practice to deploy an operating system alongside a separate integrity checking kernel.

These defenses have significantly improved the security of computer systems. Reliably mounting an attack on a modern, well-protected hardware or operating system is harder than ever. These defenses, however, are far from ideal. Existing protections against hardware-level attacks on memory systems, for example, either provide partial protection or incur high overheads. Similarly, defenses targeted at enforcing the integrity of OS kernels require complex and high overhead software stacks (such as a hypervisor and/or a separate trusted kernel), and are still prone to attacks in the presence of software bugs.

In this dissertation, we present hardware mechanisms for providing *strong* protections against an array of attacks on the memory hardware and the operating system's code and data, while incurring lower overheads compared to existing mechanisms. This chapter highlights these protections and optimizations proposed in this dissertation.

1.1 Preventing Physical Attacks on Memory Systems

Data stored in a CPU's registers and caches is extremely hard to physically probe or modify. Modern CPUs have complex layouts with multiple metal layers. As a result, it is extremely expensive, if at all possible, for an attacker to physically access on-chip storage units.

On the other hand, data stored in off-chip memory (DRAM or non-volatile memory) can be physically extracted much more easily. An attacker with physical access to a machine can probe a data bus [44], or dump memory contents through a cold boot attack [37]. For this reason, critical secrets are considered at-risk if they are stored on an off-chip memory, and applications such as disk encryption software immediately wipe out encryption keys from memory once they are no longer necessary.

Cold Boot Attacks

Even if DRAMs are expected to lose their content immediately after the system is powered off, studies have shown that they retain data for several seconds after power loss – with only a fraction of data being lost. Such data retention in DRAMs has been shown to be a security risk [107, 8, 37], as systems that rely on disk encryption and passwords often store sensitive data in DRAM under the assumption that a reboot or removal of the DRAM will destroy the data. However, in 2008, a team of researchers demonstrated that disk encryption keys could be recovered from DDR and DDR2 DRAMs by dumping out the contents of a locked computer's DRAM [37]. The contents of the DRAM can be recovered by either rebooting the locked machine into a custom OS, or by transferring the DRAM to another machine. Since charge decay in capacitors slows down significantly at lower temperatures, they cooled the DRAMs using off-the-shelf compressed air spray cans before transferring them to another machine or rebooting the locked machine. This technique came to be known as a *cold boot attack*.

Data Extraction from Non-Volatile Main Memory (NVMM). With the imminent introduction of non-volatile main memory technologies such as the Intel 3DXPoint,

extracting data from main memory is going to become significantly easier [50]. As the name implies, these technologies do not lose their content upon power loss. Furthermore, due to their increased storage capacity, NVMMs will expose more data if their content is unencrypted. Since these devices are attached to the CPU via memory buses and are accessed directly with load and store instructions, simply enabling disc encryption will not protect them.

Memory Scrambling. In recent years it has become more challenging to execute cold boot attacks due to the introduction of *memory scramblers*. Modern processors scramble data by XOR'ing it with a pseudo-random number before writing it to DRAM [63, 79]. These scramblers were introduced to mitigate the effects of excessive current fluctuations on bus lines by ensuring bits on the memory bus transition nearly 50% of the time.

Scramblers in many modern processors (*e.g.*, Intel's Skylake) have incorporated extra features that obfuscate data. Since these data obfuscation features are not necessary to mitigate the electrical problems that motivated the use of scramblers in the first place, we surmise they were added as the first line of defense against cold boot attacks.

Since the details of these scramblers remain undisclosed, it has become challenging to extract and analyze DDR3 and DDR4 DRAM contents. Although multiple attempts to replicate cold boot attacks on scrambled memory failed in the past [35, 106], recent work has demonstrated a cold boot attack that bypasses DDR3 DRAM scramblers on 2^{nd} generation Intel Core (SandyBridge) CPUs [14].

Cold Boot Attack on Scrambled Memory. The study we present in Chapter 3 of this dissertation reveals that DDR4 memory scramblers have been redesigned in Intel's 6^{th} generation CPUs in a manner that provides enhanced data obfuscation over previous generation DDR3-based scramblers. While this enhanced design is resistant to attacks that have been demonstrated in the past, we will show that it is certainly not impenetrable. In this dissertation, we reveal details of the first cold boot attack that is able to successfully extract AES keys from a scrambled DDR4 DRAM. We demonstrate this attack by extracting VeraCrypt/TrueCrypt master keys.

Low-Overhead Memory Encryption. Confidentiality of data stored in memory can be ensured by employing strong encryption. When employing memory encryption, however, we need to ensure that we do not adversely affect memory latency or bandwidth. In Chapter 3, we present analysis that shows modern high-throughput stream ciphers (e.g., ChaCha8) coupled with high-speed ASIC implementations make it practical to create strongly encrypted memories without incurring any performance penalty. We detail latency, area, and power trade-offs of memory encryption engine designs based on RTL simulation and synthesis results. As future-generation systems will utilize dense non-volatile memories, it is becoming increasingly crucial to employ strong encryption to safeguard memory contents.

Bus-Snooping Attacks

In addition to dumping memory content, attackers could probe memory buses as well. This is a more powerful attack as it makes it possible to observe values change over time. This ability to observe multiple versions of a single data can be used to defeat certain encryption schemes (e.g., stream ciphers that do not generate a unique nonce for each memory write operation). Furthermore, an attacker with the capability to modify bits on the bus can change or reset memory contents as they are being written and read.

Authenticated Memory Encryption. Stream ciphers alone are not sufficient to protect contents from tampering. To ensure data integrity, we need to use an authenticated encryption scheme that stores a message authentication code (MAC) along with the encrypted data. A MAC function is a one-way function that takes a block of data and a secret key, and produces a hash value. An attacker cannot generate a new MAC value after modifying data, as a secret key is required for generating a valid MAC.

Even if attackers cannot create MAC values for arbitrary data, they can still reset data blocks and MACs to an older value – a process commonly referred to as a replay attack. Replay attacks cannot be prevented unless we have a mechanism for detecting changes to the MACs. Such protection is provided by constructing integrity trees (discussed in the next chapter).

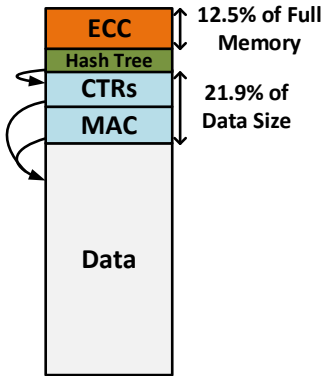


Figure 1.1 Storage Overheads of ECC & Authenticated Memory Encryption. Authenticated memory encryption requires storage of MACs and counters. Each of these incur an 11% storage overhead. The counters themselves are then protected with a hash tree. Furthermore, if the system has ECC support, an additional 12.5% storage overhead is incurred by parity bits.

Employing these cryptographic primitives in the context of memory encryption, however, results in high storage overheads as shown in Figure 1.1. Strong encryption requires that we provide a unique, one-time (non-reusable) input commonly referred to as a cryptographic nonce. Counter-mode encryption, the most widely used mode of memory encryption, uses a growing counter value as a nonce. Counters/nonces used to encrypt each memory block need to be stored as they are necessary to decrypt the data when it is read back. SGX (Intel's implementation), for example, stores a 56-bit counter for each 64-byte block, resulting in an ∼11% overhead. In addition to counter values, MACs also need to be stored for each memory block. Again, SGX uses 56-bit MACs - incurring an additional ∼11% storage overhead.

Memory read latency is also impacted as reading a protected block requires we fetch the corresponding counter and MAC values, perform decryption, and check the integrity of the data. Additional overhead arises from the fact that the counters and MACs themselves need to be protected by an integrity tree. Overall, strong memory encryption incurs more than 22% storage overhead.

Reducing the Overhead of Authenticated Memory Encryption. In Chapters 4 and 5, we propose optimizations for reducing the storage overheads discussed above. The

optimizations we present also reduce the performance overhead of authenticated memory encryption.

We make the observation that MAC bits, along with 7 parity bits, can be used for error detection and correction. As a result, the MAC storage overhead on systems with ECC memory can be reduced to zero – without forgoing security guarantees or error detection and correction capabilities. Furthermore, ECC memory has wider buses that enable the CPU to read error detection/correction bits in parallel with a memory block. Hence, this approach enables us to read MAC values in parallel with the data block – reducing the extra latency associated with reading security metadata.

We also show how delta encoding can be used to represent counters with fewer bits. Delta encoding is a data representation scheme that stores the difference (deltas) between two values, instead of storing the full values themselves. To apply this scheme, we first group multiple memory blocks into a logical group. Each block-group shares a single reference value. The counters for each block in a group are then represented as deltas to this common reference value. Since the delta values will be smaller than the actual values themselves, they can be represented with fewer bits – resulting in lower storage overhead.

1.2 Safeguarding the Operating System's Code and Data

The second part of this dissertation explores memory protection techniques for enhancing the OS kernel's security. Security mechanisms on a computer system heavily rely on the proper functioning of the OS kernel. If the kernel is compromised, other protections built on top of it can be bypassed.

Today's systems commonly employ a secure boot mechanism to prevent the system from booting into a tampered kernel. However, once the boot process is completed, attackers could exploit software vulnerabilities to perform malicious actions such as overwriting kernel memory, executing malware in kernel-mode, or disabling driver integrity checks (example attacks can be seen in [120, 19, 31, 84, 116, 101, 91, 87]).

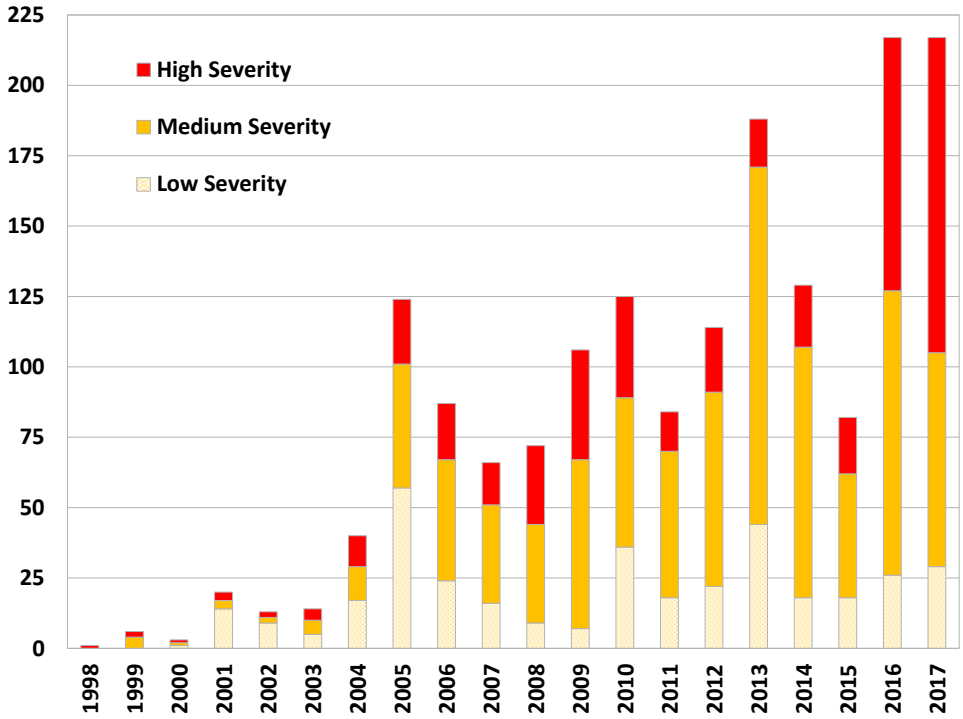


Figure 1.2 Linux Kernel Vulnerabilities. New kernel vulnerabilities continue to be reported each year (data based on the National Vulnerability Database).

Completely eliminating vulnerabilities from today's large and complex kernels remains impractical. This is illustrated by the data in Figure 1.2, which shows Linux kernel vulnerabilities disclosed since 1998 (based on the data from the National Vulnerability Database[1]). Clearly, today's operating systems have a large attack surface as a result of their size and complexity. Furthermore, kernel modules (drivers) introduce additional attack surface, as they have full access to the kernel's address space.

Ongoing Challenges of Protecting the Kernel

Recognizing their vast attack surface, a growing number of operating systems are deployed along with a continuous kernel integrity monitoring/enforcement mechanism.

[1]https://nvd.nist.gov/. Vendor: "linux", Product:"linux_kernel", Keyword: "linux kernel"

Android distributions from major mobile vendors and recent versions of Microsoft Windows are two notable operating systems that have adopted this approach [126, 16, 12].

These integrity checking mechanisms typically monitor kernel memory and CPU configuration registers to prevent malicious modifications. Furthermore, they impose additional restrictions on the system, such as disallowing writable code pages in the kernel's address space. Continuous integrity monitoring makes compromising the kernel quite challenging – even in the presence of software vulnerabilities.

Guarding the Guardians. If the integrity enforcement mechanisms run at the same privilege level as the kernel they are meant to protect, then vulnerabilities in the kernel could be exploited to subvert the integrity enforcement mechanism itself. For example, a kernel bug has previously been exploited to bypass Microsoft's Kernel Patch Protection (KPP) feature [6].

To shield these protection mechanisms from a compromised kernel, numerous previous works have proposed leveraging virtualization support available in today's systems [32, 47, 67, 114, 102, 94, 23]. With this approach, the main OS runs on top of a hypervisor, while kernel protections are implemented in a hypervisor or another *separate* virtualized operating system. This provides additional protection as the hypervisor and the virtualized operating systems are isolated by the hardware. Microsoft Windows 10 has adopted this approach to protect its driver and kernel integrity enforcement mechanisms [126].

Alternatively, it is possible to leverage the TrustZone execution environment which is available on ARM CPUs. Programs that run under TrustZone are isolated and protected from the rest of the system at the hardware level [73]. Android devices commonly take advantage of TrustZone to protect the kernel integrity mechanisms from the main kernel [12, 16].

Even with the extra hardware-level isolation, software vulnerabilities in the kernel integrity enforcement mechanisms themselves remain exploitable. For instance, attacks against multiple generations of TrustZone-based kernel protections have been shown [85, 16]. In addition, virtualization-based security relies on hypervisors, which

themselves have a large attack surface – as proven by serious vulnerabilities that continue to be discovered in hypervisors [86, 88, 89, 90]. Furthermore, running operating systems on top of a restrictive hypervisor incurs performance overheads [118, 53]

Hardware Mechanisms for Safeguarding Kernel Integrity

In the second part of this dissertation, we present an efficient hardware-based protection, dubbed Neverland, which can be used to harden the operating system against kernel-mode malware and rootkits. Neverland does not rely on a complex (and potentially vulnerable) kernel integrity monitoring software and incurs zero runtime performance overhead.

Once the kernel boots, Neverland's hardware irreversibly takes away the operating system's ability to overwrite certain configuration registers and portions of the physical address space. More specifically, we "lock" the kernel's code and read-only sections, security-critical configurations (e.g., driver signing configuration flags, system call table), and CPU configuration registers that store kernel entry points. These locked registers and memory regions can only be modified after rebooting the system. Neverland also prohibits the CPU from fetching kernel-mode code from any memory region lying outside the physical addresses assigned to the OS kernel and drivers.

Note that the code, data, and registers mentioned above are guarded or periodically scanned by a typical kernel integrity monitor. Locking these components at the hardware-level obviates the need to continuously run an integrity monitoring software.

We use a **hardware permission table** to mark portions of the physical address space as read-only, executable, privileged, and locked. On every instruction fetch and load/store, the CPU checks the permission table to determine the legality of the memory operations.

A number of embedded CPU architectures specify a form of (optional) physical memory permission tables (e.g., ARM MPU and RISC-V PMP specifications). These ISA extensions are normally used to implement a lightweight memory protection scheme in low-resource microcontrollers that do not have virtual memory support [11, 24].

CPUs with virtual memory support do not generally implement these physical memory permission schemes. The hardware support required by Neverland is a variation of such permission tables. We leverage immutable hardware tables to provide low-overhead operating system protections – without requiring complex integrity monitoring software that rely on hypervisors or trusted execution environments. In addition, we also introduce the notion of "locked" CPU configuration registers to prevent interrupt and system call hijacking.

Maintaining a separate physical address permission table has advantages over existing hardware-level privilege escalation defenses such as Intel Supervisor Mode Execution and Access Prevention (SMEP and SMAP) and ARM Privileged-eXecute-Never (PXN). SMEP/SMAP and PXN are under the total control of the operating system and can be disabled if the kernel is compromised. Neverland side-steps this issue by stripping away the operating system's ability to overwrite the permission table entries once the boot process is complete. Furthermore, since the physical address permission tables do not rely on virtual page permissions, attacks that manipulate the page table entries [84] are prevented.

We validated the efficacy of these protections by prototyping the hardware extensions in Spike (the official RISC-V emulator) and running a minimally modified version of the Linux kernel on the emulator. Our evaluations based on RTL simulation and synthesis show that the hardware extensions required by Neverland incur minimal silicon and energy cost, and that integrating the proposed protections does not incur any performance overhead beyond the 10s of milliseconds that are required to setup permissions at boot time.

1.3 Summary of Contributions

We make the following contributions towards understanding and overcoming the threat of cold boot attacks on modern memory systems:

- We demonstrate DDR interfaces continue to be vulnerable to cold boot attacks despite the introduction of increasingly advanced memory scramblers. We

demonstrate data recovery from scrambled DDR3 and DDR4 DRAMs and show how encryption keys can be extracted by descrambling memory (Chapter 3, [124]).

- We demonstrate memory scramblers in DDR4 controllers can be replaced with strong ciphers (such as ChaCha8) without introducing any performance overheads (Chapter 3, [124]).

We make the following contributions towards reducing the overhead of authenticated memory encryption schemes (which prevent bus snooping and memory tampering):

- We show how the extra memory chips and buses available in ECC DRAM can be leveraged to eliminate the message authentication code (MAC) storage overheads – without forgoing error correction or integrity checking capabilities. In addition to reducing storage overheads, this approach has the added benefit of reducing the performance impact of authenticated memory encryption by up to 15% as MACs are read in parallel with the data through the ECC bus (Chapter 4, [125]).

- We present a data encoding scheme that can reduce counter storage overhead incurred by counter-mode encryption by a factor of 7, while still enhancing encryption performance by up to 10% (as a result of reductions in off-chip memory access) (Chapter 5, [125]).

We make the following contributions towards safeguarding the operating system's memory (code/data) from kernel-mode malware and rootkits:

- We present a hardware-based memory protection scheme that can protect an operating system from kernel-mode malware and rootkits. The protection works by stripping away some of the kernel's powers once the boot process is done. We show how this protection can be applied without incurring any runtime performance overhead (Chapter 7).

- We demonstrate the practicability of the approach by integrating the proposed protections into an emulated RISC-V CPU, and by making minimal changes to the

Linux kernel to take advantage of these hardware protections. We also show the hardware and energy overheads of the additional hardware (a permission table) are minimal (Chapter 7).

Dissertation Organization. This dissertation is organized into two parts. The first part (Chapters 2 - 5) focuses on hardware-level attacks and defenses on memory systems, while the second part (Chapters 6 and 7) focuses on operating system security and our low-overhead kernel memory protection scheme. Chapter 2 and Chapter 6 provide background material for Part 1 and Part 2 of the dissertation respectively.

Part I: Efficient Physical Memory Security

Chapter 2

Physical Memory Security

Data stored in an off-chip memory, such as DRAM or non-volatile main memory, can potentially be extracted or tampered with by an attacker with physical access to a device. In the first part of this dissertation, we will focus on mechanisms for defending against such physical attacks.

This chapter provides a primer on bus snooping and cold boot attacks (two techniques that can be used to physically extract data out of main memory), and motivates the need for protecting the confidentiality and integrity of data stored in off-chip memory. We also summarize the state-of-the-art on memory encryption and authentication techniques that are employed for preventing such physical attacks.

2.1 Data Remanence and Cold Boot Attacks

DRAMs store bits by storing charge in bit cell capacitors. Due to substrate leakage, these capacitors can lose their charge in 10s of milliseconds unless the system refreshes the bit cell. For this reason, DRAMs are conventionally expected to lose their content once a system loses power. However, contrary to conventional wisdom, studies have shown that DRAM modules can maintain a large fraction of their content after being powered down. It has also been demonstrated that the bit cell capacitors can retain their charge for significantly longer periods of time (up to minutes) when the DRAM chips are super-cooled [107, 37].

This long-term retention of DRAM content poses security risks since an attacker

with physical possession of a device can move the DRAM module from a locked system to an attacker-owned machine, and extract sensitive data stored in the DRAM. In 2008, Halderman *et.al.* demonstrated that DDR and DDR2 modules can retain 99.9% of the data stored in them for minutes when they are cooled down to -50^oC using an off-the-shelf can of compressed air [37]. They exploited this fact to extract sensitive data such as disk encryption keys from locked and suspended computers – an attack vector now popularly known as a “cold boot attack".

After the demonstration of cold boot attacks, other studies have replicated the attack on additional platforms, including Android devices [65]. Another work reproduced the results from [37] and also demonstrated the feasibility of cold boot attacks on DDR3-based systems that do *not* employ any form of memory scrambling [56]. Today, many CPUs employ some form of memory scrambling that XORs data with keys generated during system boot-up. As a result, cold boot attacks have become more challenging.

2.1.1 Cold Boot Attack Mitigation Measures

To prevent extraction of encryption keys via cold boot attacks, disk encryption tools typically erase keys stored in memory immediately after a disk is unmounted. This approach can be applied on partitions other than the one the operating system is running on. While this approach reduces the attack surface, it will fail to protect disk encryption keys if a device is acquired by an attacker while disks are still mounted and the key is resident in DRAM (*e.g.*, if the machine is in sleep mode while the attacker acquires it). It should be noted that even disk encryption tools such as BitLocker that store encryption keys within trusted platform modules (TPMs) are still susceptible to cold boot attacks as the expanded keys for mounted volumes are cached in DRAM until the drive is unmounted or until the system is cleanly shutdown [98].

Solutions that store encryption keys exclusively in CPU registers have also been proposed [105, 64]. Loop-Amnesia [105] stores encryption keys in model-specific registers that are typically used by performance counters. Similarly, Tensor [64] leverages x86 debug registers for storing keys. These solutions require a patched operating system to prevent user-space access to these otherwise freely accessible

registers, as they are now storing sensitive keys. Such approaches are capable of protecting disk encryption keys, but they typically suffer performance impacts since round keys must be generated before any encryption operation and subsequently erased. Previous work has shown that expanded round keys greatly simplify the task of identifying keys in memory [37], and thus, they should not reside in memory. However, due to the lack of protected on-chip storage and the limited size of registers, a large amount of sensitive data still remains in main memory, at least for a limited time, unprotected.

Full memory encryption techniques, both in hardware and software, have been suggested [40, 122]. Furthermore, the new Intel Software Guard Extension (SGX) includes hardware support for maintaining confidentiality and integrity of data stored in DRAM by employing strong encryption (AES) and message authentication codes (MACs). Unfortunately, SGX has been shown to incur significant performance overheads [10, 117]. This makes such high-security solutions undesirable for latency-sensitive and bandwidth-intensive applications. The design of a memory encryption scheme that provides integrity at low overhead is still an open problem. Recent AMD CPUs also provide full-memory AES encryption (without integrity checking) [50].

Finally, newer machines with compact form factors come with their DRAM chips either directly soldered on the motherboard or packaged with the CPU chip together. While this can make attacks more cumbersome, it does not fully deter them. A determined attacker can still carefully desolder the DRAM modules or boot from external media (potentially after flashing the BIOS to enable boot from external media).

2.1.2 Cold Boot Attacks and Memory Scramblers

In older DDR and DDR2 systems, the CPU stores data in memory in plaintext form. This made capturing memory contents straightforward: simply transfer the DRAM to another system and read the full contents of the DRAM. With the introduction of high-speed buses, however, the scrambling of DRAM data was introduced to improve signal integrity and reduce power supply noise [63].

DRAM traffic is not random, and long successions of 1s or 0s can be observed on the

data bus under normal workloads. As a result, energy can potentially be concentrated at certain frequencies or all the data lines can switch in parallel resulting in high di/dt (current fluctuations). The noise created by these phenomena can affect signal integrity and power delivery. The Intel Core processor datasheets [80, 78, 79] state that by randomizing the DRAM data, potentially dangerous di/dt harmonics are eliminated. Consequently, the overall power demand of the bus becomes largely uniform.

Over time, however, these scramblers have been adapted to also provide data obfuscation, in particular with the introduction of scrambler seeds that change after each reboot. Another Intel product datasheet [77] states that its integrated memory controller has a *"DDR Data Scrambler to reduce power supply noise, improve signal integrity and to encrypt/protect the contents of memory.*" These data obfuscation features thwart straightforward cold boot attacks. In Chapter 3 we will present a detailed analysis that shows that modern scramblers, despite their increasing complexity, cannot prevent more advanced cold boot attacks.

2.2 Bus Snooping and Replay Attacks

A cold boot attack provides the attacker a static snapshot of the memory system. A more powerful attacker can track all DRAM read and write transactions and analyze how the data changes over time. If a memory encryption scheme does not address this threat model (e.g. a strong stream cipher used with a fixed key and nonce), the attacker can easily recover the plaintext of a specific block after collecting multiple ciphertext samples.

Data Tampering and Replay. An active attacker may also attempt to manipulate execution by flipping bits stored in memory. Such attacks cannot be prevented by solely encrypting data and require message authentication codes to be effectively prevented (discussed in the next section). An even more powerful attack involves resetting a memory block to an older version – a process commonly referred to as a "replay" attack. Replay attacks cannot be protected by simply tracking MACs as the attacker can reset

both the encrypted block and its associated MAC value. As we will detail in the next section, preventing replay attacks requires a more complex defense.

2.3 Authenticated Memory Encryption

In this section, we provide an overview of the state-of-the-art in authenticated memory encryption and the associated overheads.

Threat Model. Authenticated memory encryption aims to protect a system from an attacker that has physical access to a device. An attacker with physical access to a device can either snoop data as it is being transferred over the memory bus, or extract data directly out of DRAM modules. Authenticated memory encryption ensures confidentiality and integrity of data, but does not prevent information leakage through memory access patterns [59, 93, 109].

Early Work in Memory Encryption and Integrity Checking

Execute-Only Memory (XOM) [55] is one of the earliest efforts to design an architecture for creating tamper and copy-proof software. Under XOM, software that is to run on a specific target CPU is encrypted using the target CPU's public key. Since only the target CPU has the private key, no external entity can examine, execute, or modify the program code. As public key encryption is slow, the scheme used a symmetric cipher to encrypt the code, after which the symmetric key itself is encrypted using the public key and embedded in the program's header.

XOM's protection mechanisms do not fully prevent replay attacks – where an attacker resets contents of data in external memory to an earlier state. **Hash-tree** based memory verification was proposed to address this vulnerability [33] (detailed below). AEGIS [111] improves upon XOM by leveraging advances made in cryptography and memory integrity checking. AEGIS is resilient to replay attacks, and also reduces decryption latency through the use of **counter mode encryption** (discussed below).

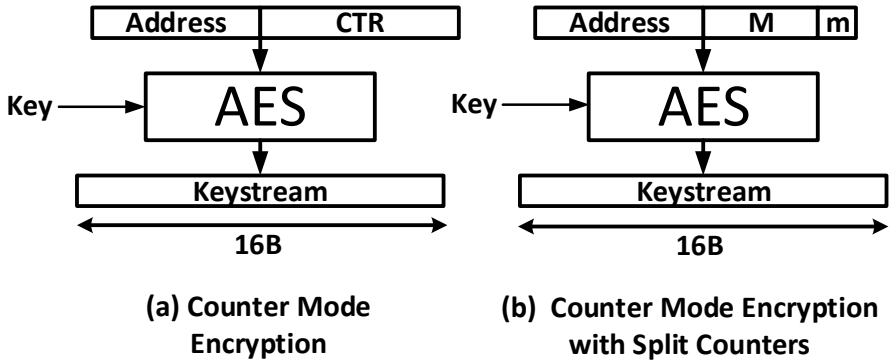


Figure 2.1 Counter (CTR) Mode Encryption. CTR mode encryption generates a keystream by encrypting a counter value with a block cipher such as AES. The counter is incremented after each encryption to avoid keystream reuse. The counter value is concatenated with the physical address of the block being encrypted to generate a unique keystream for each memory block. The generated keystream is XOR'd with the data to be encrypted/decrypted. The split counter scheme reduces storage space by storing a small (e.g., 8-bit) counter (m) per block and sharing a larger (e.g., 64 bit) counter (M) among multiple blocks.

Counter Mode Encryption.

High-performance memory encryption implementations use a block cipher (mainly AES) in counter mode. In counter mode memory encryption, a distinct counter value is associated with each 64-byte memory block. We generate a keystream for a memory block by encrypting the corresponding counter value using a block cipher such as AES. The generated keystream is then XOR'd with the data to be encrypted/decrypted (Figure 2.1a). To make the keystream unique across different memory blocks, the physical address of the memory block being encrypted is concatenated with the counter value as shown in Figure 2.1a.

To get the maximum security guarantees from CTR mode encryption, a counter value must be incremented whenever the corresponding memory block is updated. If the counters are incremented during each write, then identical values written to the same memory location at different times will appear as different ciphertexts. These counters themselves are then stored in the off-chip memory in plaintext.

Counter Storage Overhead. The size of the counters needs to be large enough to ensure they do not overflow and wrap around to 0. An overflow would result in keystream reuse – which compromises the security of the system. To prevent overflow, counters that are 64-bit or 56-bit wide are typically employed. Such large counters would not overflow for the lifetime of any machine. Using large counters, however, results in a significant storage overhead. With a 56-bit counter per 64-byte block, the counter storage overhead will be ∼11% (Figure 1.1).

Data Integrity

Securing data using counter-mode encryption alone, however, cannot prevent an attacker from modifying or resetting data. We need to provide an additional mechanism to ensure *data integrity*. Ensuring data integrity requires us to compute and store distinct message authentication codes (MACs) for each memory block. A MAC is computed using a secret key that is not known by the attacker. Similar to encryption keys, the key used to compute a MAC is securely stored in an on-chip SRAM or register.

As a DDR memory block is read/written at a 64-byte memory block granularity, MAC values are also computed and stored for each 64-byte memory block. As a result, protecting large chunks of memory results in significant storage overheads. For example, Intel SGX computes 56-bit MACs for each memory block. This results in an ∼11% overhead – in addition to the 11% counter storage overhead discussed above (Figure 1.1).

Integrity Trees

When protecting large chunks of memory, the MAC values will require more storage than what is normally feasible on-chip. For example, protecting just 128MBs of memory would require 14MBs for MAC storage (assuming 56-bit MACs per block). As a result, we need to store the MAC values in an off-chip memory.

However, when storing the MACs off-chip, we need to prevent the attacker from tampering with the MAC values themselves. If the integrity of the MAC values is not

enforced, an attacker can "replay" old values by concurrently resetting the counter, MAC, and data to an older value.

MAC values are stored off-chip in a tamper-proof manner by using **integrity trees** [33, 95, 36]. As already discussed, all data that is stored in an off-chip DRAM is not trusted. On the other hand, the amount of storage that is available on-chip is limited. The main aim of integrity trees is to protect the integrity of a large off-chip memory using a small amount of on-chip metadata storage.

Constructing an Integrity Tree. To build an integrity tree, we begin by computing a MAC value for all the data blocks that we need to protect. These MAC values (M_0) form the leaf nodes of the integrity tree. The size of the MAC bits is only a fraction of the data blocks. For example, an 8-byte (64-bit) MAC can be used to protect a 64-byte block (Intel SGX uses just 56-bit MACs [81]). We then proceed to chunk multiple MAC values together into a single memory block. If our MACs are 8 bytes, we can chunk 8 of them together and store them in a single memory block.

To protect M_0 from tampering, we compute yet another set of MACs (M_1) over the memory blocks that are storing M_0. The M_1 MACs form the second level of the tree. Note that M_1 will be smaller than M_0. We repeat this process recursively until we reach a level where the MACs can fit in an allocated on-chip memory. Since the top level is stored in an on-chip memory, it cannot be easily tampered with.

Reading/Writing Protected Data. When reading a memory block, its MAC value is computed and checked against the value stored in its parent node. This check is then recursively applied until we reach the root node. Since the attacker cannot tamper with the root node (which is stored on-chip), any changes made to the nodes of the tree can be detected during the final check made against the root node. Similarly, when a data block in memory is modified, all of its parent nodes need to be updated.

2.4 Previously Proposed Optimizations

Multiple optimizations have been proposed to reduce the performance and storage overhead associated with encryption and integrity checking. In this section, we highlight prior optimizations that are relevant to our work [33, 95, 121].

Counter and MAC Caches. Verifying data integrity by recursively reading nodes from the integrity tree requires extra memory reads. To reduce the overhead incurred by these extra reads, Gassend et al. cache the integrity tree on an on-chip cache [33]. Just like conventional CPU caches, these caches reduce the latency for reading MACs and counters – especially when the accesses exhibit spatial and temporal locality. Intel's SGX implementation has a dedicated cache for MACs and counters [36, 81].

Bonsai Merkel Trees. The work by Rogers et al. [95] made notable enhancements to tree-based integrity checking. They made the observation that it is possible to ensure the integrity of data blocks by only protecting the integrity of counters. Since the size of counters is significantly smaller than the size of data blocks, protecting the counters (instead of the data) results in a significantly smaller tree – which the authors call a Bonsai Merkle tree.

Their technique requires the counters used for encryption to also be used as an additional input when computing MAC tags for the data blocks. With this modification, if an attacker changes the data or MAC values in memory without changing the counter value, the integrity checks will fail. In short, modifying data without detection requires modifying the protected counters as well. Intel SGX uses this optimized tree structure [36, 81]. We also use Bonsai Merkel trees as the baseline in our evaluations and apply our proposed optimization over them.

Split-Counters. Yan et al. [121] proposed **split counters** – a compact counter storage scheme (Figure 2.1b). Split counters reduce the overall size of counters by storing small (8-bit or less) minor counters (m) per memory block. However, an 8-bit counter would easily overflow, resulting in nonce re-use after just hundreds of writes to the

same memory block. Re-encrypting the entire memory with a new key when a single counter overflows would be extremely expensive.

The authors address this issue by coupling minor counters with a 64-bit major counter (M) – which would not overflow for millennia. A single major counter is shared by multiple consecutive blocks, incurring significantly less storage overhead compared to storing a 64-bit counter for every block. The consecutive blocks that share a major counter form a block-group [1] – which is typically a few kilobytes. When a block is accessed, it's minor counter is concatenated with its major counter to obtain the full counter value (Figure 2.1b). When a minor counter overflows, the entire block-group is re-encrypted using a new major counter. This enables the technique to avoid re-encrypting the entire memory.

Split counters can reduce the counter storage overhead by a factor of 8 compared to storing a 64-bit counter for each memory block. However, this counter compaction scheme requires frequent re-encryption of block-groups on memory intensive applications. In Chapter 5, we present a counter encoding and storage scheme that results in a significantly lower rate of re-encryption while maintaining the compactness of split counters.

[1] The original paper refers to block-groups as "pages". We avoid using that terminology throughout this dissertation to avoid confusion with OS pages.

Chapter 3

Cold Boot Attacks on Scrambled DDR3 and DDR4 Memory

As we have described in the previous chapter, previous work has demonstrated that systems with unencrypted DRAM interfaces are susceptible to cold boot attacks [37, 65, 14]. This method has been shown to be an effective attack vector for extracting disk encryption keys out of locked devices.

However, most modern systems incorporate some form of data scrambling into their DRAM interfaces [14, 76] making cold boot attacks challenging. While first added as a measure to improve signal integrity and reduce power supply noise, these scramblers today serve the added purpose of obscuring the DRAM contents. As a result, multiple attempts to perform cold boot attacks on systems with scrambled DRAM have failed in the past due to lack of detailed information about the scrambler design [35, 106]. This has led some to conclude that scrambled memory systems are not susceptible to cold boot attacks [35, 38].

In this chapter, we will begin by presenting an empirical analysis of the data obfuscation features found in (older) Intel DDR3 memory scramblers and then investigate enhancements that have been introduced in the newer DDR4 memory scramblers. Our study reveals that DDR4 memory scramblers have been redesigned in Intel's 6^{th} generation Core (Skylake) processors in a manner that provides enhanced data obfuscation over previous generation DDR3-based scramblers. While this enhanced design is resistant to attacks that have been demonstrated in the past, it is certainly not impenetrable. In this work, we reveal details of our DDR4-based cold boot attack that is

able to successfully extract AES keys from a DDR4 DRAM connected to an Intel Skylake processor. We demonstrate this attack by extracting VeraCrypt/TrueCrypt master keys.

The limitations of memory scramblers we point out in this chapter motivate the need for strong, yet low-overhead full-memory encryption schemes that can transparently replace scramblers. Existing schemes such as Intel's SGX can effectively prevent such attacks, but have overheads that may not be acceptable for performance-sensitive applications [10, 117]. To that end, we present analyses that confirm modern stream ciphers such as ChaCha8 are sufficiently fast that it is now possible to completely overlap keystream generation with DRAM row buffer access, thereby enabling the creation of strongly encrypted DRAMs with zero exposed latency. Adopting such low-overhead measures in future generation of products can effectively shut down cold boot attacks in systems where the overhead of existing memory encryption schemes is unacceptable. Furthermore, the emergence of non-volatile DIMMs that fit into DDR4 buses is going to exacerbate the risk of cold boot attacks. Hence, strong full memory encryption is going to be even more crucial on such systems.

Our goal in this work is not to criticize the state of memory scramblers, but to make two important observations: *i*) DRAM (including DDR4) continues to be susceptible to cold boot attacks as the scramblers do not provide sufficient confidentiality guarantees, and *ii)* modern high-throughput stream ciphers (*e.g.*, ChaCha8, CTR mode AES-128) coupled with high-speed ASIC implementations make it practical to create strongly encrypted memories that are impervious to cold boot attacks without incurring any performance penalty.

3.1 Overview of Memory Scramblers

In older DDR and DDR2 systems, the CPU stores data in memory in plaintext form. This made capturing memory contents straightforward. With the introduction of high-speed buses however, the scrambling of DRAM data was introduced to improve signal integrity and reduce power supply noise [63].

DRAM traffic is not random and successive 1s and 0s can be observed on the data

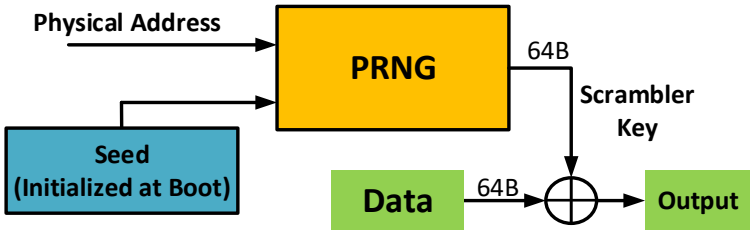


Figure 3.1 High-level View of Memory Scrambling. Data is scrambled by XOR'ing it with a pseudo-random number. The scramble/descramble process is symmetric and a seed (generated at boot time) and portions of the physical address bits are used by the pseudo-random number generator (PRNG) to generate 64-byte keys.

bus under normal workloads. As a result, energy can potentially be concentrated at certain frequencies or all the data lines can switch in parallel resulting in high di/dt (current fluctuations). The noise created by these phenomenon can affect signal integrity and power delivery. The Intel Core processor datasheets [80, 78, 79] state that by randomizing the DRAM data, potentially dangerous di/dt harmonics are eliminated. Consequently, the overall power demand of the bus becomes largely uniform.

Over time, however, these scramblers have been adapted to also provide data obfuscation, in particular with the introduction of scrambler seeds that change after each reboot. Another Intel product datasheet [77] states that its integrated memory controller has a *"DDR Data Scrambler to reduce power supply noise, improve signal integrity and to encrypt/protect the contents of memory.*" These data obfuscation features thwart straightforward cold boot attacks.

Figure 3.1 provides a high level model of a data scrambling unit. It is very similar to a symmetric encryption scheme. Before data leaves the CPU, it is XOR'd with pseudo-random numbers. When data is read back from DRAM, it is XOR'd with the same pseudo-random number to recover the original data. The pseudo random number generater (PRNG) can potentially be initialized using a certain initial seed.

While Intel's datasheets do not provide any additional details about their data scrambler architecture, their 2011 publication [63] discloses that Linear Feedback Shift Registers (LFSRs) are used as pseudo-random number generators (PRNGs) by

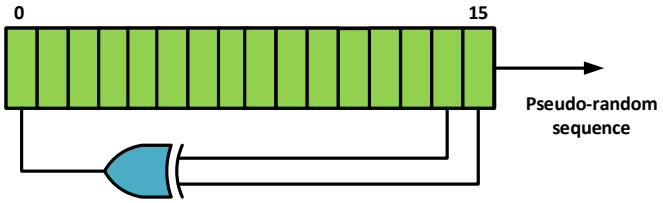


Figure 3.2 An Example Linear Feedback Shift Register

the scrambler implemented in the Westmere microarchitecture. An LFSR is a simple hardware component commonly used to generate pseudo-random numbers. It consists of a shift register and a feedback function that sets the leftmost bit of the shift register. The feedback function is conventionally an XOR of some of the bits of the shift register. Figure 3.2 shows an example PRNG constructed out of an LFSR. The specific design shown in the figure is the standard LFSR used to scramble data on one of the buses on a hybrid memory cube [76] . Different random number sequences can be generated by varying the initial state of the LFSR, register width, and the bits that are XOR'd together. The Intel publication [63] also discloses that the LFSRs are seeded using a portion of the address bits. This reduces correlations between memory blocks containing the same data values.

3.1.1 Memory Scramblers and Cold Boot Attacks

The information presented so far gives us a high-level understanding of the inner workings of memory scramblers. However, this information is not sufficient to analyze to what extent scramblers can defend against cold boot attacks. In the rest of this chapter, we will present an empirical analysis of multiple generations of memory and show that these interfaces continue to be vulnerable to cold boot attacks.

CPU Model	Microarchitecture	Launch Date
Intel i5-2540M (DDR3)	SandyBridge	Q1, 2011
Intel i5-2430M (DDR3)	SandyBridge	Q4, 2011
Intel i7-3540M (DDR3)	IvyBridge	Q1, 2013
Intel i5-6400 (DDR4)	Skylake	Q3, 2015
Intel i5-6600K (DDR4)	Skylake	Q3, 2015

Table 3.1 CPU Models of Tested Machines. In this work, we analyzed the DDR3 and DDR4 memory scramblers of the listed processors.

3.2 Analysis Framework

Since the scramblers implemented in modern CPUs are not publicly documented, we needed to empirically analyze the data transformations applied by the memory controller before attempting to identify its limitations. For this study, we analyzed data stored by the DDR4 memory controllers integrated in Intel's 6^{th} Generation Core Processors. For comparison purposes, we also analyzed older scramblers found in multiple generations of DDR3 controllers. We performed this analysis on multiple laptop and desktop computers. The CPUs we have analyzed are given in Table 3.1.

All data that is eventually written to DRAM passes through the scrambler. Similarly, all data that is read by software is first passed through the descrambler and regular software cannot see the raw scrambled data. This scrambling/descrambling algorithm is implemented inside the memory controller, which cannot be directly accessed. Hence, we needed to devise a mechanism for capturing and observing the raw output of the memory scrambler. We did this using two approaches.

For the DDR4 DRAMs, we relied on a motherboard that enabled us to switch the scramblers on and off through the BIOS configuration menus. However, the DDR3-based systems we used for comparative analysis do not expose a mechanism for controlling the scrambler. Hence, we relied on an external FPGA-based system to directly access the DDR3 memory's contents. On the FPGA board we can read and write any raw (unscrambled) data. For our experiments, we used the Xilinx VC709 board with Virtex-7 FPGA to write unscrambled data to the DRAM (Figure 3.3).

To extract the scrambler keys, we implemented a "reverse cold boot attack" on

a memory filled with all zeros. We use the mechanisms we just described to write *unscrambled* zeros to a DRAM module. Given that the final step of scrambling is XOR'ing the scramble key with the data, we can discover the keys by initially filling all memory with *unscrambled* zeros and then re-reading the data with the scrambler turned-on. In this case note that when the zeros are read back through the descrambler, it will attempt to descramble the data using the scrambler keys and we are actually reading the scrambler keys themselves (*i.e.*, $0 \oplus key$). Based on this approach, we extract the scrambler keys using the following steps:

1. On a system where scrambling is disabled, we fill the entire memory with raw (unscrambled) zeros.

2. We freeze the DRAM and transfer it onto the motherboard of the system we are analyzing.

3. We boot scrambled system and read the raw zero values from memory using our custom GRUB module that runs on the bare hardware.

The resulting memory image retrieved by the program (a GRUB module) is filled with scrambler keys (since a scrambler key XOR'd with zero yields the key). The program we run to extract the memory dump has no operating system or virtual memory manager running underneath it. Hence, we have full view of DRAM contents while introducing minimal pollution to the memory contents. Note that this procedure is the reverse of a cold boot attack, since in this situation we want to inject known data *into* a scrambled system.

Instead of filling the DRAM with zeros, we can alternatively begin by allowing the DRAM to fully decay to its ground state. We can then read out the value each DRAM block assumes at this ground state *with the scrambler turned off*. Note that portions of the DRAM cells decay to a zero while others decay to a one. After this initial "profiling" stage, we can boot into a scrambled system with the fully decayed DRAM and read out this known data (*i.e.,* the ground state values) through the scrambler. Unlike the technique where we fill the memory with zeros, we do *not* have to worry about bit decay that might occur in midst of the experiment.

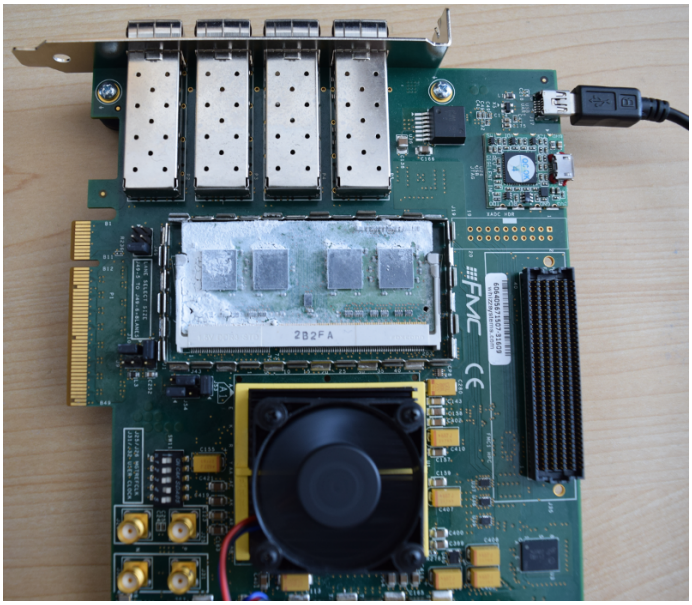


Figure 3.3 FPGA Used for Initializing DDR3 DRAM with Zeros.

Later in our research, we acquired a DDR4-based motherboard that allowed us to reboot an initially scrambled machine with the memory scramblers turned off – without destroying the scrambled DRAM contents from the previous boot cycle. Hence we were able to study the data transformations made by the scrambler by simply writing scrambled data to memory and reading it back out on the next boot cycle with the scrambler turned off. It should be noted that this setup was used to speed up our analysis, and the cold boot attacks detailed later in this section were indeed tested by transporting a frozen DDR4 DRAM across two machines. Figure 3.4 shows the frozen DDR4 DRAM on the scrambled machine's motherboard, prior to being pulled out and re-socketed into the motherboard of a machine with a disabled scrambler.

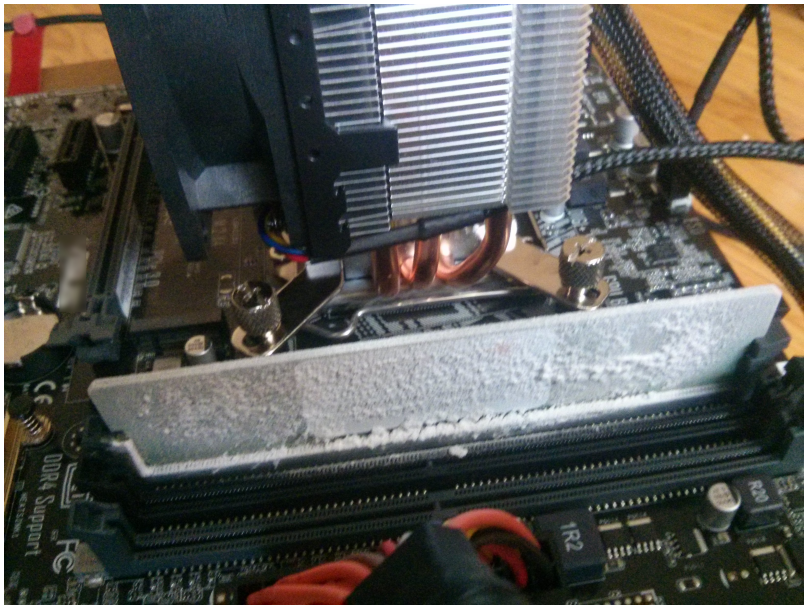


Figure 3.4 Cold Boot Attack on DDR4 DRAM. This photo shows the DRAM in one of our DDR4-based systems. The DRAM is filled with data scrambled by the memory interface of an Intel Skylake-based CPU. The memory has been cooled to $-25°C$, and it will next be moved to a separate system where its contents will be descrambled.

3.3 Analysis of DDR3 Scramblers

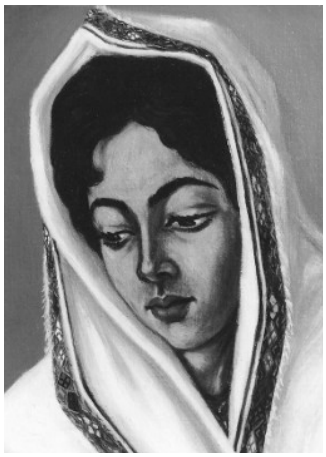
In this section, we present an empirical analysis of DDR3 scramblers based on the experimental setup we described in the previous section [1].

3.3.1 Seeding the Scrambler

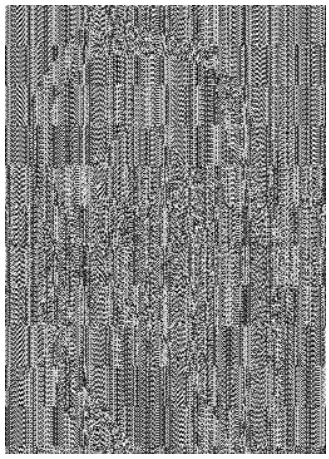
One property that can be seen by observing the list of keys used for scrambling is that the keys change every time the system reboots. Keys extracted for one boot session cannot be used to descramble memory contents on future boot sessions. This make cold boot attacks more challenging.

When studying the BIOS firmware for an Intel SandyBridge machine [2], we observed

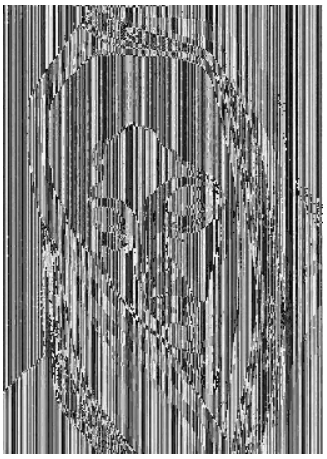
[1] A concurrent work by Bauer et al. [14] also analyzes Intel's DDR3 scramblers. The analysis in dissertation extends into the more advanced DDR4 scramblers as well.



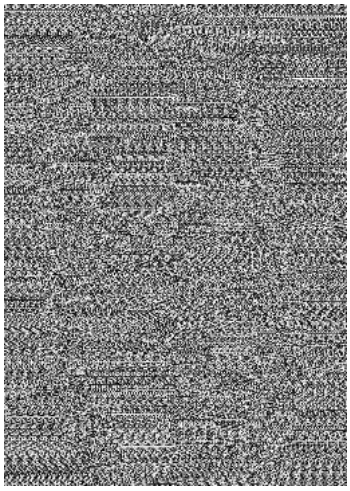
(a) Original Image



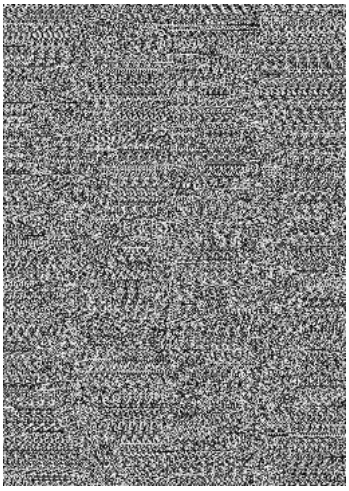
(b) Scrambled DDR3 Data



(c) Scrambled DDR3 Data Read Back After Reboot



(d) Scrambled DDR4 Data



(e) Scrambled DDR4 Data Read Back After Reboot

Figure 3.5 Visual Comparison of DDR3 and DDR4 Scramblers on Single Channel Machines. Due to the larger key pool used in the Skylake DDR4 scramblers, repeated data in memory reveal fewer correlations compared to DDR3 (compare (b) and (d)). Additionally, unlike DDR3, portions of the key are not factored out in the DDR4 scramblers when data is loaded back using a different seed (compare (c) and (e)). Overall, DDR4 memory achieves better data obfuscation.

the memory scrambler is initialized with a 32-bit random seed during the boot process. The BIOS can change this seed on each reboot. This seed is used by the scrambler's psedudo-random number generator. Details on the seed generation process for the particular firmware we analyzed can be found in the Appendix.

3.3.2 Memory Scrambler Properties

The aforementioned 32-bit random seed is not directly used to scrambler memory blocks. On DDR3 and DDR4 systems, data is read in blocks of 64 bytes. Therefore, the 32-bit seed must be "expanded" into a 64-byte random number that can be XOR'd with the entire memory block (see Figure 3.1). Using the techniques we described in Section 3.2, we extracted the 64-byte scrambler keys that are directly XOR'd with the data. Our observations on these 64-byte scrambler keys can be summarized as follows:

1. A limited number of scrambler keys are **reused multiple times** to scramble the whole memory. We observed a total of 16 scrambler keys per memory channel.
2. On a reboot, **all scrambler keys change** based on the new seed.
3. The mapping of which memory blocks share the same scrambler key is fixed and does not change after reboot.
4. Each memory channel has its own set of distinct 16 scrambler keys

The fact that addresses from different channels are scrambled using keys specific to that channel suggests that the scrambler's output also depends on a portion of the address bits. This is consistent with a disclosure from Intel [63].

Figure 3.5(b) illustrates how the raw data shown on 3.5(a) is scrambled and stored by the DDR3 controller. While the image in Figure 3.5(b) is obscured due to scrambling, features of the original image are still present. This correlation resulted from the scramblers reuse of a single key multiple times. Clearly, this reuse of keys can be leveraged to descramble data in a straight forward manner.

Key Idea: A limited number of keys are reused multiple times to scramble a large memory space. The assignment of which memory blocks use the same keys is fixed and does not change after reboot.

Reading Data After Reboot

As mentioned above, the scrambler keys are changed after every reboot. As a result, we will not get the original data written to DRAM if we try to re-read it after a cold-reboot.

To get additional insights into the scrambler's characteristics, we analyzed how data re-read after system reboot differs from the original one. Figure 3.5(c) illustrate how the original data in Figure 3.5(a) would appear when we read it back after reboot. Perhaps surprisingly, when the original data is read back after reboot, it will appear as though the data that has been scrambled using just a single key per channel! In other words, every memory block in a channel can be descrambled by XOR'ing them with a single, fixed 64-byte value. Note that a total of 16 scrambler keys (per channel) are used to originally scramble the data. However, simply rebooting the system and re-reading the data will make it appear as though a single scrambler key was used for each channel.

To understand what is happening, suppose data D_1 is stored at address A_1 after being scrambled using key K_1. Similarly, assume data D_2 at address A_2 is scrambled using key K_2. The memory content at the two addresses will be:

- $A_1 : D_1 \oplus K_1$
- $A_2 : D_2 \oplus K_2$

When we read data back from memory after reboot, it will be descrambled using different keys (K_1' and K_2') and we will *not* get the original data. Instead, data read from the two addresses will be:

- $A_1 : (D_1 \oplus K_1) \oplus K_1'$

- $A_2 : (D_2 \oplus K_2) \oplus K_2'$

For any two pair of memory locations we picked, we observed the following relationship holds:

$$K_1 \oplus K_1' = K_2 \oplus K_2' \tag{3.1}$$

The implication is that re-reading the data from DRAM after reboot is equivalent to reading out data encrypted using a single key. Since all the 64-byte memory blocks are encrypted using a single 64-byte key, this scheme resembles a block cipher operating in electronic codebook (ECB) mode.

Based on the relationship in Equation 3.1, we can infer the two following properties:

- A portion of the scrambler key generation process does not depend on the memory address, as Equation 3.1 holds true for any two arbitrary address pairs.
- The part of the scrambler key that depends on the address is "factored out" on reboot, as the operation $K_i \oplus K_i'$ is constant for any memory location i.

Based in these inferences, we can model the scrambler as being composed of two pseudo random number generators (PRNGs). One PRNG is a function of the seed generated at boot and the other PRNG is a function of the physical address(A).

The scrambler can hence be represented as:

$$K(A, S) = F(A) \oplus G(S), \tag{3.2}$$

where F and G are pseudo-random number generators (PRNGs), S is the scrambler seed, and A is memory address.

Across cold reboots, the only thing that changes in Equation 3.2 is S. Figure 3.6 is a graphical depiction of the DDR3 memory scrambler model based on the observations presented this far. While this model gives us a simple way to represent the scrambler, note that it may not be how the scrambler is actually implemented in hardware.

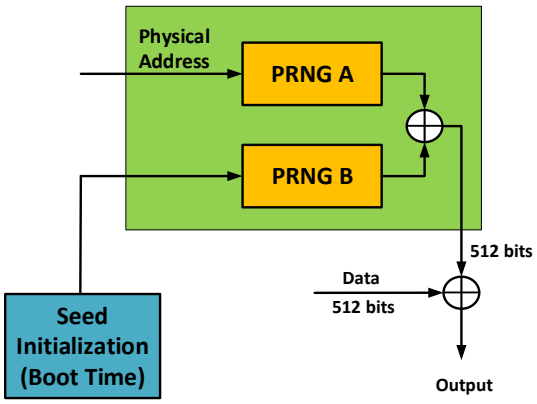


Figure 3.6 A Model Representing Scrambler Logic

Key Idea: On a single channel DDR3 system, re-reading data after reboot is analogous to reading data scrambled with a single key. This resembles a block cipher operating in electronic codebook (ECB) mode.

Dual-Channel Machines

Most modern systems have multiple semi-independent memory controllers (channels) that can access multiple memory modules independently. The previous section analyzed the operation of the memory scrambler on a single-channel configuration. We now extend our analysis to a dual-channel system.

After reboot, unlike the single channel case (where data appeared to have been encrypted with the key $G(S) \oplus G(S')$, data appears to have been scrambled with two distinct keys - one key for each channel. On systems with a SandyBridge CPU, the lower five bits are used as block offset and the sixth bit of the address is used to pick a channel to access [69, 99]. Hence, after system reboot, memory blocks will appear to have been encrypted with two keys in an alternating fashion.

On the other hand, newer micro-architectures XOR multiple bits together to pick a channel [99]. For example, a recent study[69] discloses that, on a dual-channel

Ivy-Bridge machine, bits 7, 8, 9, 12, 13, 18, 19 of the physical address are XOR'd to select a channel. On the IvyBridge machine we tested (i7-3540M) we were able to verify that memory addresses that map to the same channel will appear as being scrambled with identical keys after reboot. Despite these added complexities, it is still possible to descramble a multi-channel memory, as long as the channel mapping is known, the scramble key mapping is known, and sufficient memory contents are known in advance.

Key Idea: On dual-channel machines we examined, reading data after cold reboot is analogous to reading data scrambled with two separate keys – one key per channel.

3.3.3 Effects of Descrambling on a Separate Machine

If we can reset a machine and boot from an external media, the observations made thus far are sufficient to descramble memory contents. However, in the event that booting from an external media is disabled using various mechanisms (BIOS password, secure boot), attackers can potentially perform cold boot attacks by removing and placing the DRAM in another machine. We access the feasibility of such attacks under three distinct cases.

1. **Descrambling on an Identical CPU Model:** We transferred a DRAM module across two machines with an i7-3540M (IvyBridge) CPU. The results we had were identical to the case where the DRAM was read on the same machine after a cold reboot.

2. **Descrambling on a CPU with Different Model Number, but Identical Microarchitecture:** Intel releases different CPU models under the same micro-architecture. We transferred a DRAM module across two single-channel machines that have different models of SandyBridge CPUs (i5-2430M and i5-2540M). Cold boot attacks with this configuration were successful, as the final memory contents appeared to have been scrambled with 16 distinct keys. This result suggests that

the scrambler architecture does not change in any significant way across CPU models with the same microarchitecture.

3. **Descrambling on a CPU with a Different Microarchitecture:** A recent study has shown that CPUs with different microarchitectures use different algorithms to map a physical address to DRAM locations[69]. The specific DRAM location pointed by address *A* in one machine is different from the DRAM location pointed to by the same address on another machine. Consequently, this configuration is not acceptable for cold boot attacks unless the differences in the scramble key generation and memory mapping could be discovered.

Key Idea: Reading a scrambled DRAM using a separate but identical system yields the same result as reading memory on the same machine after reboot. On the other hand, different generations of CPUs implement different random number generation and memory mapping algorithms. Hence, descrambling data on a separate machine with a non-identical CPU poses additional challenge and is not preferred.

3.4 Analysis of a DDR4 Scrambler

We now present our analysis of DDR4 scramblers, and how they differ from the previous generation scramblers in DDR3 memory controllers.

After analyzing the extracted keys and their characteristics throughout the memory and between subsequent boots of the system, we were able to make the following observations for the DDR4 memory scramblers in Intel's Skylake CPUs:

- A memory channel is scrambled using a total of 4096 distinct 64-byte keys (in contrast to just 16 keys in the DDR3-based memory systems). While visible correlations could exist for the same data in different 64-byte blocks, their probability of occurrence compared to DDR3-based DRAM is reduced by a factor of 256. This effect can be seen by comparing Figures 3.5b and 3.5d.

- These 4096 keys generated for every channel are all reset after system reboot. However, BIOS from certain vendors do not reset the scrambler seed every boot cycle and the same set of scrambler keys are reused after reboot.
- Unlike older DDR3-based scramblers, reading back data on an identical machine after reboot (*i.e.*, after the scrambler is reset) does not result in the entire memory being scrambled with a single 64-byte key. This can be seen by comparing Figures 3.5c) and 3.5e). That is, the XOR of all the corresponding current keys and the previous keys does not result in a single universal 64-byte key. As such, cold boot attacks devised for scrambled DDR3 DRAM are not applicable to Intel Skylake based DDR4 systems as attacks on DDR3 relied on discovering a single 64-byte universal key.
- Similar to the DDR3 systems, the scrambler keys appear to be generated using a combination of a scrambler seed generated at boot time by the BIOS and portions of the physical address bits. Consequently, different memory blocks that share a scrambler key continue to share a scrambler key after reboot.

To descramble a DDR4 DRAM during a cold boot attack, we need a mechanism to recover the scrambler keys solely from data captured out of a scrambled DRAM. Since a zero value XOR'd with the scrambler key will result in the key itself, memory blocks with zeros written to them will contain the actual scrambler keys. It has been shown that zeros occur more frequently than most other individual values in memory – an occurrence which has been a basis for multiple proposed memory compression algorithms.

Therefore, the challenge lies in identifying which memory blocks contain scrambler keys (*i.e.*, are zero'd memory blocks). Previous attacks on DDR3 systems only had to extract one key for each channel and hence relied on straightforward frequency analysis [14]. However, due to the large number of keys at play in the newer systems, we cannot reliably use simple frequency analysis.

The key to identifying a scrambler key in a memory dump lies in an observation that we made regarding properties of the scrambler keys. After extracting the scrambler keys using the technique detailed above, we were able to identify *invariants on the scrambler*

keys that we used to form a *scrambler key litmus test*. These litmus tests allowed us to identify zero-filled blocks in memory images that reveal a scrambler key. The invariants are between byte pairs in a 64-byte scrambler key.

These invariants are better understood by partitioning the 64-byte memory block into 2-byte words. In the expressions below, *K[i:j] represents bytes within a 64-byte scrambler key starting at byte i and ending at byte j*.

Using this notation we can describe relationships that hold true within any **64-byte scrambler key**:

$$K[i:i+1] \oplus K[i+2:i+3] = K[i+8:i+9] \oplus K[i+10:i+11]$$

$$K[i:i+1] \oplus K[i+4:i+5] = K[i+8:i+9] \oplus K[i+12:i+13]$$

$$K[i:i+1] \oplus K[i+6:i+7] = K[i+8:i+9] \oplus K[i+14:i+15]$$

$$for\ i = 0, 16, 32, 48\ (i.e.,\ for\ each\ \textbf{16-byte aligned words})$$

While it is possible to setup a system of boolean equations using the above expressions and attempt to find candidate solutions for the unscrambled text, we have found that approach to be computationally intensive. Instead, we use these expressions as a litmus test to check if a given memory block in a true DDR4 memory dump is a likely 64-byte zero-value block (thus being a scrambler key exposed in the memory dump). Even on a heavily loaded system, we were able to mine all scrambler keys by running the tests on less than 16MB of the memory dump. Consequently, a small memory dump can quickly produce all of the keys used. These litmus tests are still valid and can extract keys required for descrambling even when data is read back through a scrambler with a different set of keys. As a result, an attacker does not require a machine with a disabled scrambler.

It should be noted that portions of the bits stored in the DRAM can decay while the DRAM is being transported to the attacker's machine. We will discuss how we tolerate such data loss in the next subsection.

Key Idea: The DDR4 scrambler generates 4096 distinct scrambler keys *for each channel*. These keys can be mined from a memory dump by testing memory blocks against a set of litmus tests. These tests can be performed in a manner that is resilient to modest bit flips.

3.4.1 Disk Encryption Key Recovery from a DDR4 Memory

We now turn our attention to designing a cold boot attack on a Skylake-based DDR4 system. In this attack, the scrambled memory dump is obtained by extracting a frozen DDR4 DRAM from the secure system, and placing it in a system with a disabled scrambler where it can be dumped to disk. The proof-of-concept attack we present here focuses on recovering the AES encryption keys, specifically those used to decrypt a secure TrueCrypt/VeraCrypt disk volume on a Linux machine. However, it can be extended to extract any other information.

Attack Model The attack we present here assumes the attacker has no knowledge of which memory blocks share the same scrambler key, and the attacker has no specific knowledge of the unscrambled contents in the scrambled memory. These assumptions helps to demonstrate that simple permutations of the random number generators and key mapping schemes (as different generations of DDR3 controllers have done in the past) would not affect this attack's ability to recover sensitive information. If a second machine is used for dumping the memory image (instead of rebooting the same machine), then the attacker must use a CPU that is the same generation as the one being attacked. This restriction is important as different generations of Intel CPUs can have different physical address to channel, rank, bank, and row mappings. As noted above, the scrambler on the attacker's machine or on the machine being attacked does not need to be turned off when capturing memory images.

Like previous cold boot attacks on unscrambled memory systems (*e.g., DDR and DDR2*), we search for an expanded AES key, which has special properties that allow it to be easily distinguished from all other data in the system [37]. Our search, however,

is complicated by the fact that an AES round keys can span four 64-byte memory blocks, thereby requiring us to guess four different scrambler keys from a total of 4096 possibilities (8192 for a dual channel system) to fully descramble the key table. If brute forced, this would result in 2^{48} different combinations for each set of four memory blocks on a single channel system. To work around this limitation, we modified the algorithm in [37] to recover AES keys from a scrambled memory without having to descramble more than a single 64-byte block at a time. Fortunately (for the attacker, and unfortunately for all else) we can test if a given 64-byte memory block contains portions of the AES round keys. Thus, we can form an *AES key litmus test* for a 64-byte memory block that, if it holds true, tells us if we are in middle of contiguous memory blocks that contain AES round keys.

Specifically, our attack algorithm works as follows on a scrambled DDR4 memory dump:

1. Scan the memory image for 64-byte aligned, zero-filled memory blocks that reveal scrambler keys directly. These candidate keys, K, are located when they pass the *scrambler key litmus test* detailed in previous section. Note that not all of the candidate keys K are scrambler keys. However, many of them are and those that occur more frequently are likely keys.

2. Using the candidate scrambler keys, K, gathered in the previous step, descramble individual memory blocks in the dump with all keys K, looking for descrambled memory blocks that pass the 64-byte block *AES key litmus test* (explained in detail below).

3. For all descrambled memory blocks that pass the 64-byte block AES key litmus test (S_i, K_j), repeat Step 3 on neighboring blocks until a complete set of AES round keys have been located.

4. When a complete set of AES round keys has been found, recover the secret AES key from the head of the table.

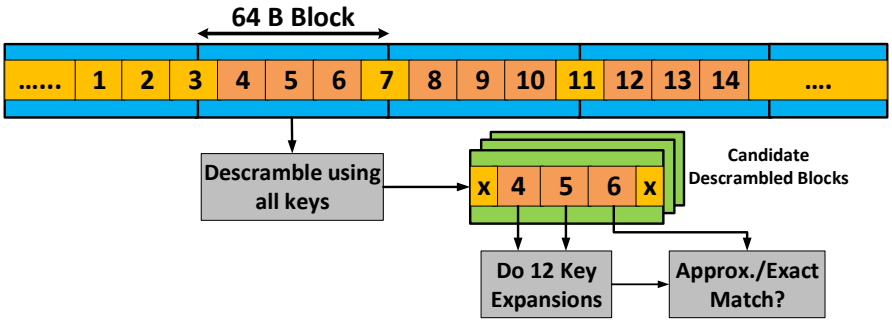


Figure 3.7 Scanning Memory for AES Round Keys. Our attack locates the AES round keys used to decrypt the disk. To locate the round keys, we descramble a 64-byte block using all (thousands) of the candidate scrambler keys. If the block contains portions of AES round keys, it will be possible to successfully run at least one iteration of the AES key expansion algorithm (since three round keys fit in 64 bytes). Since we do not know which three round keys lie in the block, we need to try all 12 possible expansions (*e.g.*, 1,2,3 and 2,3,4, *etc.*).

AES Key Litmus Test The standard AES algorithm can operate with a key length of 128, 192, or 256 bits. However, the key supplied to the algorithm is *expanded* to form a longer key using an algorithm that only depends on the key. This expansion is necessary since the algorithm encrypts data by applying a round function multiple times, using a different key each time. For example, in AES-256, a 256-bit key is expanded to generate 16-bit keys for each of the 14 rounds – forming a total of 240 bytes. These round keys are normally computed once and stored in memory.

The AES key search algorithm described in [37] works by sliding a search window across a stream of bytes looking for an expanded AES key. However, this algorithm assumes the full memory image is descrambled ahead of time. As a result, it picks 256-bits of data (for AES-256) and applies the standard key expansion algorithm. Similar to their algorithm, we rely on the contiguous storage of round key in memory for recovering keys. However, we do not require the memory image to be fully descrambled for the algorithm to work. Our modified algorithm is based on one straightforward insight: in a contiguous memory region containing AES round keys, at least 3 consecutive round keys will reside in a 64-byte memory block, regardless of how the key is aligned in memory. An example is shown in Figure 3.7. Except the first memory block, all the

others contain 3 full round keys (*e.g.*, the second memory block in the figure contains complete keys for rounds 4, 5, and, 6). If the data structure storing the key happens to be aligned to 64 byte boundaries, 4 round keys would end up in a single block. For all other cases, however, 3 of the keys would appear unfragmented in a single memory block.

Due to this guarantee, it is possible to check if a single descrambled memory block is storing portions of an expanded encryption key. We first create descrambled candidate blocks by XOR'ing a scrambled memory block with all the candidate scrambler keys. Then, for each candidate descrambled block, we take 256 bits of data (with varied offsets) and pass it through the key expansion algorithm. Since we do not know which round keys we are going to encounter, we cannot simply apply the standard key expansion algorithm. Instead, we do all 12 possible partial expansions for AES-256 by executing the key expansion algorithm starting at each of the 12 different rounds. The expansion results are then checked against the stream of bytes adjacent to the 32 bytes we just expanded. The fact that multiple contiguous blocks will pass this check when an expanded key is encountered enables us to be resilient to bit decay that might have occurred while acquiring the memory image.

Once we encounter a series of contiguous memory blocks containing AES round keys, we check blocks at the boundaries to extract any remaining bytes that are part of the key. In Figure 3.7 for example, bytes from keys for rounds 1 and 2 need to be extracted from the memory block that appears immediately before the group of memory blocks we have identified. This step might not be necessary depending on the alignment of the data structure. By performing this scan on the memory dump, we were able to successfully extract AES-256 keys. For AES-128 and AES-192, we can run the same algorithm using their respective key expansion algorithms.

Tolerating Data Loss. Due to the possibility of bit decay while transferring cooled DRAM, in all the algorithms described above, we measure hamming distance to test equality instead of relying on a simple bit-by-bit comparison. Additionally, since a single scrambler keystream appears multiple times inside a memory dump, we are able to filter out modest bit flips with minimal effort.

Attack Performance. Our implementation speeds up the search process by leveraging the Intel AES instruction set extensions (AES-NI). AES-NI provides us with hardware support for performing fast key expansion. Using this algorithm we were able to scan 100MBs of memory using a single core in just 2 hours. Furthermore, since the task is fully parallelizable, we can analyze gigabytes of data in a matter of hours using multiple machines. For example, using a machine with an eight-core Intel Xeon D1541 CPU, we are able to fully search an 8 GB DDR4 DRAM image in just over 21 hours.

3.4.2 Physical Characteristics of DDR4 DRAM

DRAM modules manufactured today are much denser than the DRAMs originally attacked in [37]. To assess the feasibility of cold boot attacks on today's denser and smaller components, we measured the retention time of five DDR3 and two DDR4 modules from various manufacturers. At normal operating temperatures, a significant fraction of the data is lost within 3 seconds of losing power. To measure retention characteristics at reduced temperatures, we sprayed the DRAM with an off-the-shelf compressed gas duster to super-cool them. The super-cooled the DRAMs reached a temperature of approximately $-25°C$. In all cases, we observed that the modules are capable of retaining 90%-99% of their charges if transferred to another machine in approximately 5 seconds after being unplugged from a live system. Interestingly, one of the DDR3 modules we tested leaked data faster than the newer DDR4 modules. The algorithms we presented in this work are resilient to these modest bit flips.

It should be noted that DRAM manufacturers cannot reduce the "volume" of capacitors beyond a 10s of femto Farads without compromising reliability or significantly increasing the DRAM refresh rate (which has remained fixed over many previous generations of DRAM). For this reason, we believe that DRAM modules will continue to be susceptible to cold boot attacks for the foreseeable future. More importantly, the emergence of non-volatile DIMMs that fit into DDR4 buses is going to exacerbate the risk of cold boot attacks. Hence, strong memory encryption is going to be more crucial on these systems.

3.5 Replacing Scramblers with Strong Ciphers

Our results demonstrate that current memory scramblers cannot provide meaningful protection against cold boot attacks since they use PRNGs that are not cryptographically secure. On the other hand, replacing memory scramblers with cryptographically strong cipher engines (*e.g.*, ChaCha, AES) can provide significantly better protection against cold boot attacks, since any cold boot attack would require brute-force decryption of the strong cipher. Both strong encryption and scrambling aim to transform data into highly random bit streams. Hence, cipher engines will also mitigate the electrical problems that led to the initial introduction of memory scramblers (see Section 2.1.2). By definition, a secure encryption algorithm is indistinguishable from randomly generated data, which is the desirable characteristic of data being transmitted on a high-speed bus.

Encrypting memory contents is going to be even more important in the near future due to the imminent adoption of dense non-volatile RAM (NVRAM) DIMMs [5]. These DIMMs are being designed as a stand-alone storage or as a hardware managed backing store for DRAMs. In either case, these emerging memory technologies can hold many secrets, and the attacker would not even need to cool down the modules before transferring data to a separate machine.

3.5.1 Low Overhead Memory Encryption

In this section, we argue that power-efficient cipher engines can be used to transparently replace memory scramblers in commodity processors without incurring any performance overhead. While the encryption scheme we analyze here cannot prevent bus snooping and memory replay attacks, it is sufficient for preventing any form of cold boot attack.

Encryption Schemes. We consider two candidate ciphers to replace memory scramblers: AES and ChaCha (8, 12 and, 20 rounds). AES has been the standard cipher for most applications and hardware vendors already have hardware IP for it, making it an attractive candidate. On the other hand, ChaCha20 [17] is gaining popularity due to its strong security guarantees and higher throughput on systems that do not provide

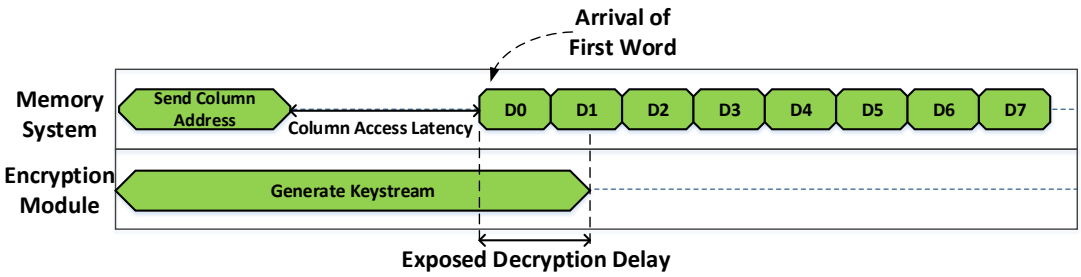


Figure 3.8 Minimizing Decryption Overhead. The key to minimizing memory cipher overheads is to avoid serializing memory access and cryptography and to instead overlap cryptography with memory access. Stream ciphers (*e.g.*, AES CTR) make it is possible to generate the keystream in parallel with accessing memory. If the keystream generation completes within the time required to transfer data from a DRAM row buffer (*i.e.*, the fastest DRAM access), there will be no exposed latency for strongly encrpyted memory. Our analyses show that there are modern crypto engines that are indeed fast enough to have zero exposed latency.

AES hardware acceleration. The fact that a pure software implementation of Chacha runs faster than a software implementation of AES has made it very attractive for mobile devices. In fact, for the past two years, nearly 100% of HTTPS connections between Android versions of Chrome and Google have been using ChaCha20 [21]. Two alternative ciphers with a reduced number of rounds, ChaCha8 and ChaCha12, have also been designed for use in systems that are willing to forgo the extra security margins provided by ChaCha20 in return for reduced computational complexity and further increased throughput [17]. Although there are numerous fast stream ciphers that have been proposed in the past, we do not consider them here as they have not undergone the rigorous public cryptanalysis that AES and ChaCha has endured.

AES-CTR and ChaCha operate as counter-based stream ciphers, permitting us to perform keystream generation without having the corresponding plaintext or ciphertext. Instead of encrypting the block directly, these ciphers encrypt an incrementing counter, which is then XOR'd with the plaintext to produce the ciphertext. This mode of operation is particularly attractive for our application because decryption could proceed in parallel with DRAM access. Before we delve into the hardware design trade-offs, we describe how the ciphers were setup.

- **AES:** We use AES in counter mode, with the physical address as a counter, and with a nonce[2] and a key generated at boot time. A memory block in DDR3 and DDR4 is 512-bits, which is four times the size of an AES block. To encrypt a memory block we need to generate four key streams using four different counter values. Since the hardware module can be pipelined, it is possible to generate the four key streams using a single hardware module with only one cycle of delay between encryption/decryption of each 16-byte blocks.

- **ChaCha:** Similar to the above scheme, we use the physical address as a counter, along with a key generated at boot time. In addition to a counter, the ChaCha cipher requires a separate nonce. For this nonce, we also rely on the availability of a boot-time random number generator.

Threat Model and Security Guarantees. The above scheme uses a fixed nonce and counter for repeated writes to a single memory block. However, each memory block is encrypted using a unique nonce or counter. This results in the following guarantees and weaknesses:

- Cold Boot Attacks: Since a unique counter value is used for each memory block, an attacker looking at a single snapshot of memory will see memory blocks encrypted using different keystreams. No memory correlation will exist and decrypting memory without knowledge of the AES key will be intractable.

- Bus-Snooping Attacks: An attacker that is able to monitor the memory bus can observe multiple reads and writes to the same memory block. And since the nonce for a given physical address is fixed, the attacker can acquire multiple blocks encrypted using the same nonce and counter. Consequently, an attacker could replay these recorded blocks without detection, thus, our approach does not protect the system against bus replay attacks. More capable technologies such as Intel's SGX can prevent such attacks at the cost of reduced performance [36, 7].

[2] A nonce is an input value that is not supposed to be used more than once. A unique nonce is typically generated for every encryption/decryption operation.

Minimizing Encryption Overhead. The most straight forward way to encrypt/decrypt bus transactions is to perform the keystream generation when data arrives in the memory controller. The main problem with this approach is that it introduces unacceptable delays on memory reads. Delays on memory writes are tolerable as the CPU can proceed with other tasks while stores are being performed. It is crucial that we reduce decryption delays since memory read latency is one of the major bottlenecks in today's systems.

Multiple works in the past have explored schemes to overlap cryptographic computations with memory reads [122, 127, 110, 121, 59, 93, 121]. One way to reduce the overhead of decrypting memory reads is to overlap the process of keystream generation with data transfer on the bus. Figure 3.8 shows the final portion of the memory read process in the DDR protocol. After a row has been read into the row buffer, the memory controller sends column access (CAS) signals. The amount of time it will take for the DRAM module to place the requested columns on the bus is deterministic and fixed for the specific DRAM module.

We leverage the deterministic time window that is available between a DRAM read request and a response from DRAM to hide the overhead incurred by memory encryption. This time window can be used to perform keystream generation, which runs independent of the data for both AES in counter-mode or ChaCha. If the entire keystream generation can be completed within this time window, then the CPU will not experience any delays for implementing fully encrypted memory.

Analyzing the Impact of Full Memory Encryption. To quantify the time window available for key expansion, we looked at the timing characteristics of DDR4 DRAM modules. According to the DDR4 standard there are only 9 allowable column access latencies that manufacturers are allowed to target. All of these standard column access latencies are between 12.5ns and 15.01ns [4]. We use these numbers as a basis for measuring exposed latency due to strong encryption. Implementations of alternative memory standards such as the Hybrid Memory Cube (HMC) have even higher transfer latency in return for higher throughput SerDes links [108].

To evaluate the performance overhead, we must know the keystream generation delay for the ciphers. To assess this delay we ran RTL simulation and synthesis on

efficient AES and ChaCha implementations. Our design exploration for AES is based on a modified version of an open-source design [43]. We used the Synopsis Design Compiler to synthesize the designs to the 45nm IBM silicon-on-insulator (SOI) technology library. Since this is a trailing edge technology, the results we generate will be slightly pessimistic compared to what a design might achieve in a newer silicon technology. However, we expect the comparisons we make below with respect to older 45nm CPUs to hold true for newer silicon technology since both the encryption pipeline and the CPUs will scale in a similar manner.

Hardware Design Trade-offs. Depending on different design decisions, the encryption modules can be optimized for latency, throughout, or low-power operation. Here, we detail the different design decisions we made.

Speed vs Area and Power. Both AES and ChaCha apply the same round function multiple times on a block of data. This gives us the option to have a single hardware unit for a round function and time-multiplex it. Such design will result in lower throughput, but also lower power. In addition, high-performance memory controllers can have multiple outstanding requests. For this reason, it is advantageous to chain multiple instances of the hardware units for the round function. In the designs we evaluated, we have dedicated units for each round. These units are then pipelined for increased throughput with multiple outstanding requests.

AES Pipeline Stages. AES rounds can be implemented with lookup tables, and this makes them amenable for faster designs. The design we used for this evaluation was adapted from [43], and it implements the sub-byte, shift row, and mix column steps as register look ups. The deeply pipelined design in [43] takes 2 cycles per round, and it is capable of running at 2.5 GHz in 45nm silicon, providing a maximum throughput of 40 GB/s. However, we chose to pipeline the design in a way that only takes 1 cycle per round, thereby slightly reducing its maximum clock frequency to 2.4GHz which reduces throughput to 39 GB/s. This slight reduction in throughput enabled us to lower the latency of generating a 16-byte key stream from a counter by about 50%.

Cipher	Maximum Freq.(GHz)	Cycles per 64B	Maximum Pipeline Delay (ns)
AES-128	2.4	13	5.4
AES-256	2.4	17	7.08
ChaCha8	1.96	18	9.18
ChaCha12	1.96	26	13.27
ChaCha20	1.96	42	21.42

Table 3.2 Cipher Engine Performance (45nm). This table provides the speed of the five cipher engines analyzed. All implementations were synthesized to a 45nm silicon-on-insulator technology. The latencies presented here do not include potential queuing delays.

ChaCha Pipeline Stages. Implementing a ChaCha quarter round in hardware requires a chain of 32-bit adders and XOR gates. In our design, we broke a quarter round into 2 pipeline stages. This enabled us to clock the design about 2 times faster (at 1.96 GHz) relative to a design where a quarter round is a single pipeline stage. This increased the frequency and resulted in a modest reduction of the latency. As we will outline in our results, this frequency enables the encryption engine to keep up with high-speed buses.

3.5.2 Results and Discussion

Cipher Engine Performance. Table 3.2 presents the performance characteristics of the synthesized cipher engines. We can see the latency that would be incurred by these cryptographic modules is not acceptable unless it can be hidden by overlapping the key generation with DDR4 DRAM column reads. Since any DDR4 module would take at least 12.5ns for a column access with a row buffer hit, AES-128, AES-256, and ChaCha8 seem like viable alternatives. The numbers also suggest that AES-128 would have lower latency when even compared to ChaCha8. However, there is an advantage to using ChaCha8 under higher bandwidth utilizations. Since AES operates on 16-byte blocks (as opposed to 64-byte blocks in ChaCha), we need to load 4 counters into the pipeline for each 64-byte memory block. This property of AES can become a disadvantage when there are numerous row buffer hits on a single channel (*i.e.*, under high bandwidth utilization).

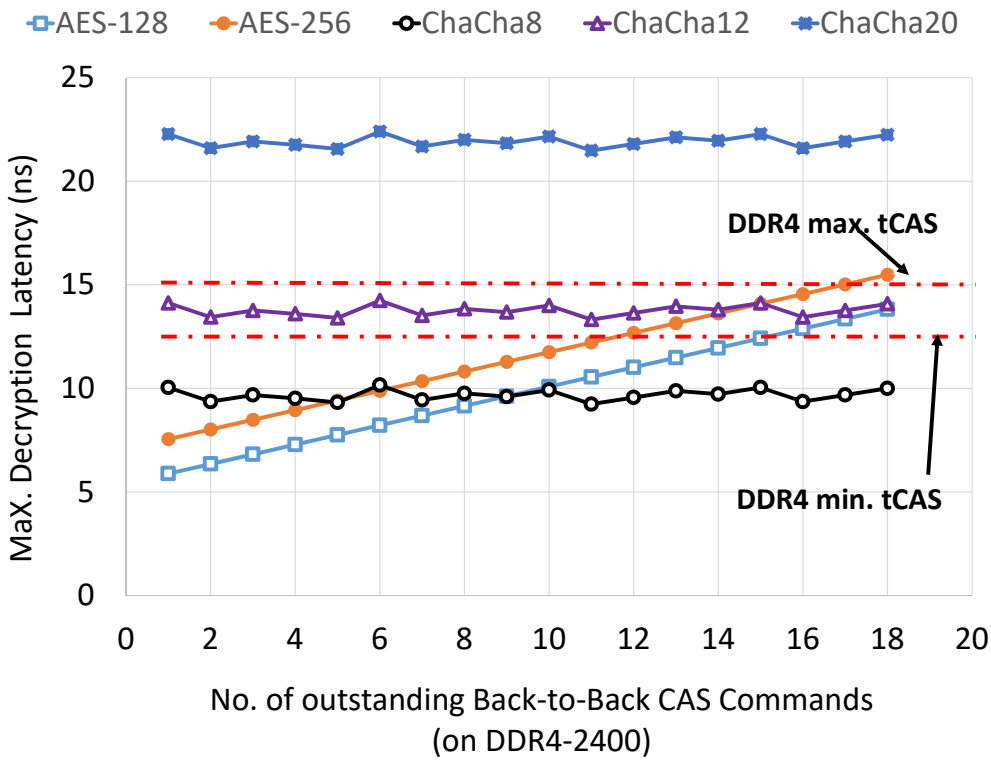


Figure 3.9 Decryption Latency of Different Ciphers. ChaCha8 is able to complete decryption faster than the minimum DDR4 read delay (12.5 ns), thus, there would be no exposed latency on encrypted DRAM reads under all loads. At lower bandwidth utilization (*i.e.*, fewer back-to-back reads with different keys), AES exhibits better performance. However, as the bandwidth utilization approaches its peak, the queuing delay starts slow AES, while ChaCha8 continues to perform well.

To analyze the performance of the cipher engines under high bandwidth utilization, we simulated the performance of the modules under different loads. Higher bandwidth utilization occurs when there are multiple row buffer hits across different banks. In the DDR4 standard, even if we might have dozens of banks on a channel, the total number of outstanding CAS commands will ultimately be limited by the contention on the bus. With a fast DDR4 module running at 1.2GHz (DDR4-2400), we can theoretically have up to 18 back-to-back CAS requests, provided that there are enough row buffer hits.

Figure 3.9 graphs the performance of the cipher engines at varying levels of memory

bandwidth utilization for a DDR4-2400 module. Note that the standard CAS latencies under DDR4 all lie between 12.5ns and 15.01ns. When the number of outstanding requests is low, AES-128 and AES-256 show superior performance. However, as the number of outstanding requests increase, the queuing delay at the input of the AES modules starts to affect the latency. As mentioned earlier, this results from the need to feed 4 counter/nonce values into the AES pipeline for every column read operation. On the other hand, ChaCha produces a 64-byte keystream from a single counter/nonce. And since this module can be clocked at least as fast as any DDR4 bus, there will be no queuing delays incurred.

The results show that ChaCha8 and AES-128 are the most suitable ciphers for replacing memory scramblers. ChaCha8 is able to complete decryption faster than the minimum DDR4 read delay under all loads. AES-128 would also have zero exposed latency except when subjected to excessive outstanding CAS requests. Even under maximum outstanding back-to-back CAS requests, AES-128 would only have a worst case exposed latency of 1.3ns.

Power and Area Overhead. To understand the power and area overhead of replacing scramblers with strong cipher engines, we compare the size and power consumption of ChaCha8 and AES-128 modules against various Intel cores. We perform a technology neutral comparison, as both the cores and cipher engines are implemented in 45nm silicon. Additionally, technology scaling is unlikely to change these results, since both the cores and cipher engines would scale in a similar manner. We make power and area comparisons against 45nm Intel CPUs: the Atom N280 (mobile), Core i3-330M (desktop), Core i5-700 (high-end desktop), and Xeon W3520 (server) CPUs. We used the power profiles and die size values stated on their respective product sheets. The power and area results are presented in Figure 3.10. We assume that there is one encryption module per-channel for each of the comparisons. As a result, we multiplied the power and area numbers of a single encryption module by the number of channels in the system.

We used the Synopsis Design Compiler for power (static and dynamic) and area estimation. For estimating the dynamic power, we used signal activity factors under full

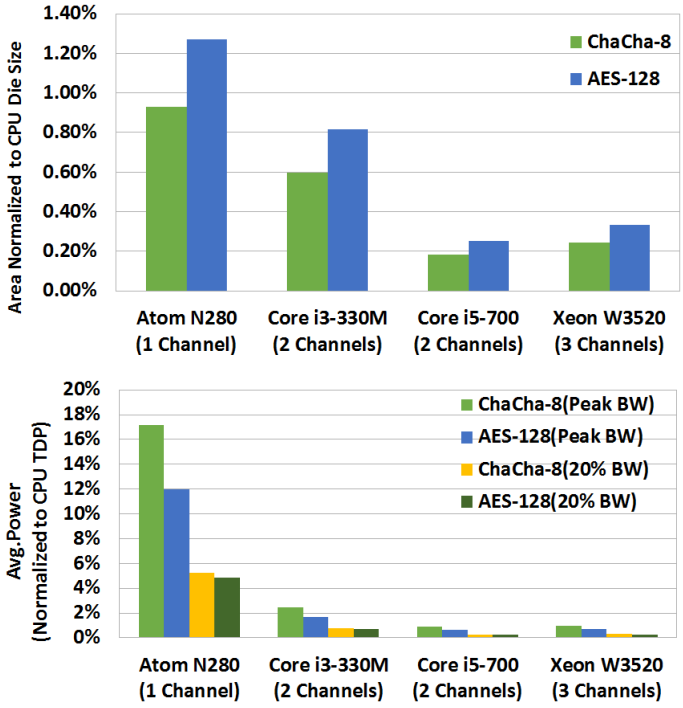


Figure 3.10 Power and Area Overhead. This figure gives estimated area and power overheads for multiple platforms, ranging from a low-end CPU (Atom N280) to a high-end server (Xeon W3520). Area overheads are uniformly low, and power overheads for the larger more capable cores are also low. The Atom CPU overheads grow, but are mitigated for lower channel utilization.

bandwidth utilization, where back-to-back CAS requests are generated whenever the bus is free. As previous work [29] has shown that most workloads utilize only a fraction of DRAM bandwidth, we also present power overheads for 20% utilization by scaling down the dynamic power to 20% of the maximum dynamic power. The analysis in [29] shows that even data intensive applications such as media streaming only use up to 15% of DRAM bandwidth which makes our estimates at 20% bandwidth utilization conservative.

Clearly, the overall power and area overheads for strong encryption are very low. In all cases, the area overheads are about or below 1%, with the expected slightly higher overheads on the small Atom CPU. The power overheads are all below 3%,

except for the single core Atom CPU, which experiences up to a 17% power increase under full bandwidth utilization. This is to be expected due to the greatly increased energy efficiency of the Atom CPU. Under more realistic workloads, however, the power overhead of the Atom CPU is estimated to be below 6%.

For low-power mobile devices, more energy-efficient memory encryption can be achieved by using cipher engines that have much lower performance than what we proposed here. Such trade-off is possible as mobile-CPUs are not likely to produce a large number of back-to-back CAS requests as server-grade CPUs and co-processors can potentially do.

Key Idea: Memory scramblers can be replaced with strong stream ciphers such as ChaCha8. For such low overhead ciphers, the process of keystream generation, which is independent of the data being encrypted, can be fully overlapped with DRAM row buffer access – thereby completely hiding the overhead of data decryption during memory reads.

3.6 Summary

With the introduction of memory scramblers in modern processors, cold boot attacks have become more challenging, as attackers must first descramble the contents of DRAM. In this work, we demonstrated that the weak data obfuscation afforded by scramblers can be readily overcome. We develop and demonstrate a straightforward means to descramble DDR3 and DDR4 DRAM connected to an Intel CPUs by exploiting the data correlations that are created due to the reuse of a limited number of scrambler keys. We presented a cold boot attack that is able to extract AES keys (including VeraCrypt/TrueCrypt master keys) from scrambled memory. Finally, we show hardware encryption performance results that suggest that memory scramblers could be readily replaced with strong stream ciphers without incurring any performance overhead. We show that ChaCha8 can fully overlap decryption with the row buffer reads in a DDR4 DRAM module, leaving no exposed latency for strongly encrypted DRAM. Similarly,

we show that the power overheads for implementing a strongly encrypted DRAM are quite low. Given the increasing size of memories and the introduction of non-volatility, memories are prone to holding more secrets for longer periods of time. As such, it is becoming increasingly important to protect the contents of system RAM.

Chapter 4

Leveraging ECC DIMMs for Reducing the Overhead of Authenticated Memory Encryption

The previous chapter focused on the confidentiality of data stored in main memory. However, the stream ciphers discussed above are not sufficient for tamper-resistant storage, and as detailed in 2.3, an authenticated memory encryption scheme is required to detect tampering.

To ensure data integrity, a message authentication code (MAC) needs to be stored for every protected memory block. Intel SGX, for example, stores a 56-bit MAC for each 64-byte memory block – which imposes an $\sim$ 11% storage overhead (see 2.3 for more details). Furthermore, an extra memory transaction is needed to access these MACs.

In this chapter, we present a solution that leverages standard ECC DRAM to reduce the overhead of integrity checking – without forgoing the error detection and correction capabilities ECC DRAMs are meant to support. Merging integrity checking (authentication) and error correction will benefit systems that already use ECC DRAM. All servers deployed by major cloud providers today are fitted with ECC DRAM [71, 30, 42]. In addition, even high-end desktops and laptops with ECC DRAM can be purchased on the market today.

4.1 Merging ECC and Integrity Checking

ECC-capable DRAM stores Hamming error correction codes along with data words. Current mainstream ECC implementations store 8 extra bits for every 8-byte memory word, resulting in a 12.5% storage overhead. With this overhead, it is possible to achieve single-bit error correction and double-bit error detection (SEC-DED) within an 8-byte word.

While regular DRAM channels have 64-bit wide data buses, DRAM modules and channels that support ECC have 72-bit wide data buses. This enables memory controllers to read the ECC bits in parallel with the information bits. As a result, the memory controller is able to perform independent error checks for each 64-bit bus transaction.

We propose using the extra storage and bus reserved for ECC bits to store MACs – *without forgoing DRAM error detection and correction*. Merging integrity checking and ECC in this manner provides significant storage savings. SEC-DED ECC incurs a 12.5% storage overhead, while 56-bit MAC tags incur an additional 11% storage overhead. When a system employs both error correction and tamper-proof DRAM storage, these storage overheads add up to consume around $1/4^{th}$ of the DRAM space covered by integrity checking and ECC (note that the MAC bits themselves need to be protected using ECC bits). Merging ECC and integrity checking reduces this overhead to a total of 12.5% – the storage overhead of employing ECC only.

Merging ECC and integrity checking also enables us to avoid the extra DRAM transaction required for reading/writing MACs, as these MAC bits will be accessed in parallel with the 64-byte data block. Furthermore, as MAC bits stored as ECC bits will immediately be available whenever we read a data block from DRAM, we do not need to cache the MAC values along with the counters – thereby freeing up on-chip counter cache space.

While implementing this scheme, however, we do *not* want to lose the error detection and correction capabilities afforded by ECC DRAM. As we will detail in the remainder of this section, MACs can be used for powerful error detection and limited error correction when hardware faults occur. Previous work [45] has shown how hashes/MACs and full integrity trees can be used for error detection and correction. While the brute-force

error correction algorithm we present below bears some similarity to [45], our main focus here is to enhance the performance of memory integrity checking while still providing double-bit error correction.

4.2 On the Security of SGX's 56-bit MAC Tags

The ECC scheme we propose here relies on the 56-bit MAC tags as introduced by Intel SGX. While 56-bit MACs are typically considered short for security purposes, it has been shown that they provide sufficient security guarantees in the context of memory encryption.

To understand the security guarantees provided by 56-bit MACs, [36] presents a security analysis of the SGX memory encryption engine. They assume an unrealistically powerful attacker who can collect cipher text samples, or repeatedly attempt forging a MAC (by instantly rebooting the machine on a failed attempt) at a rate that is beyond what is practical. They point out that, other serious practical limitations aside, the rate of MAC forgery is bounded by the throughput of the hardware under attack. They estimate successfully forging a 56-bit MAC would take 2 million years.

Hence, in this work we will assume the use of 56-bit MAC tags for enforcing the memory integrity. We also assume the MAC tags are generated using the custom Carter-Wegman MAC employed by Intel SGX [36, 81].

4.3 MACs for Error Detection

One component of ECC is error detection. MACs can easily be used for *error detection*. If a bit in a DRAM block is flipped as a result of a hardware fault, the MAC tag checks will fail. This check is part of the standard integrity checking mechanism.

Corrupted MACs However, we need to address one major issue before effectively using MACs for error detection. Hamming codes protect both information bits and the

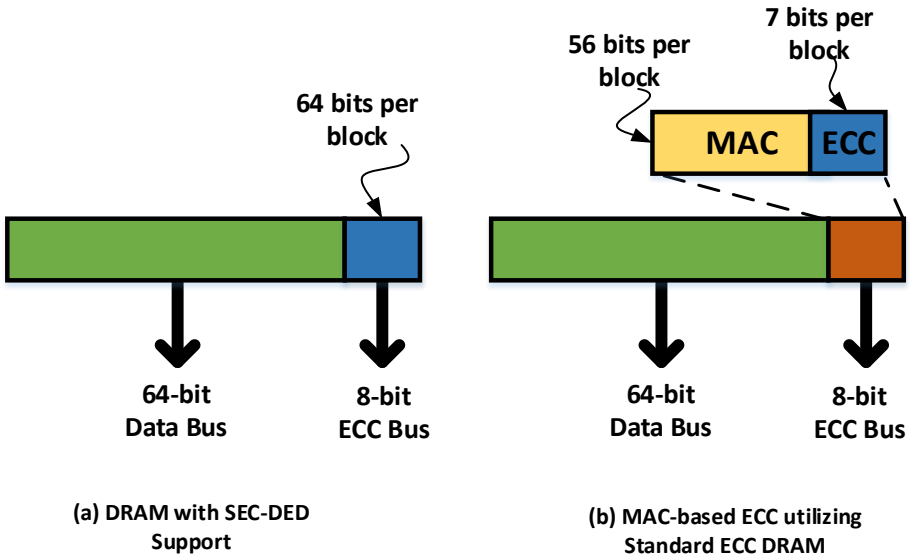


Figure 4.1 MAC-based ECC Using ECC DRAM DIMMs. ECC DRAMs are fitted with additional chips for storing 64-bit parity per 64-byte block, and have extra buses for reading parity bits in parallel with the data. We enable efficient error correction and authentication by storing 56-bit MACs and 7-bit parity in the space reserved for 64-bit parity. The MACs are used for authentication, error detection, and error correction. The extra 7-bit parity provide ECC for MAC tags.

additional ECC bits themselves. In contrast, if a MAC check fails, we cannot determine whether it's the MAC or data bits that are corrupted. As we will explain below, we need to store extra bits along with MAC tags to address this issue.

On standard ECC DRAM, a total of 64 extra bits are reserved for ECC per 64-byte memory block. Out of the 64 bits normally reserved for ECC, we use 56 bits for storing MAC values computed over the data. These 56 bits provide robust error detection for the data blocks – provided there are no bit-flips in the MAC bits themselves.

To detect and correct bit-flips in the MACs themselves, we generate ECC bits for protecting the 56-bit MACs. If we use the standard SEC-DED scheme, we only need 7 ECC bits to detect double-bit errors and correct single-bit errors in the MAC itself. With these additional ECC bits, we now have 56 bits reserved for MACs and 7 bits reserved for ECC, amounting to a total of 63 bits. These bits fit in the space reserved for storing 64 ECC bits for every block (Figure 4.1).

	Error Detection	Error Correction
SEC-DED ECC	Up to 2 bit-errors per 8-byte word	1 bit-error per 8-byte word
MAC-Based ECC	Full error detection	Expensive multi-bit correction

Table 4.1 Comparison of SEC-DED and MAC-based ECC. MAC-based error detection outperforms SEC-DED whereas the effectiveness of error correction under the two schemes depends on the location of the bit-flips. Example scenarios are given in Figure 4.2.

Comparison with Standard Error Detection Standard ECC memory (which uses Hamming codes) can detect up to 2 bit-flips per 8-byte words. On a full 64-byte memory block, we can potentially detect up to 16 bit-errors – provided that we do not have more than 2 bit-flips per 8-byte word (Table 4.1).

MAC-based error detection has a different property. On the MAC bits themselves, we will have 2-bit error detection (as we use standard SEC-DED to protect them). On the data bits, however, we have full error detection, i.e., any number of bit-flips can be detected. Figure 4.2 provides some examples which illustrate the effectiveness of the two schemes under different faults on the cipher text (excluding the MAC).

Enabling Efficient Scrubbing The MAC and parity bits occupy 63 out of the 64 bits that are available for storing ECC bits. We use the remaining 1 bit for storing a single parity bit computed over the cipher text. This bit can be used by a DRAM scrubbing hardware/firmware (which typically rely on parity bits) to quickly and efficiently scan for single-bit errors without re-computing MACs. The hamming coded MACs can also be scrubbed as hamming codes contain a parity bit.

4.4 MACs for Error Correction

The more challenging aspect is utilizing MACs for error correction. For single-bit errors, Hamming codes employed in SEC-DED tell us which bit flipped. MAC tags, however, only tell us there is a bit flip – but not which bit(s) flipped. As a result, error correction becomes more challenging.

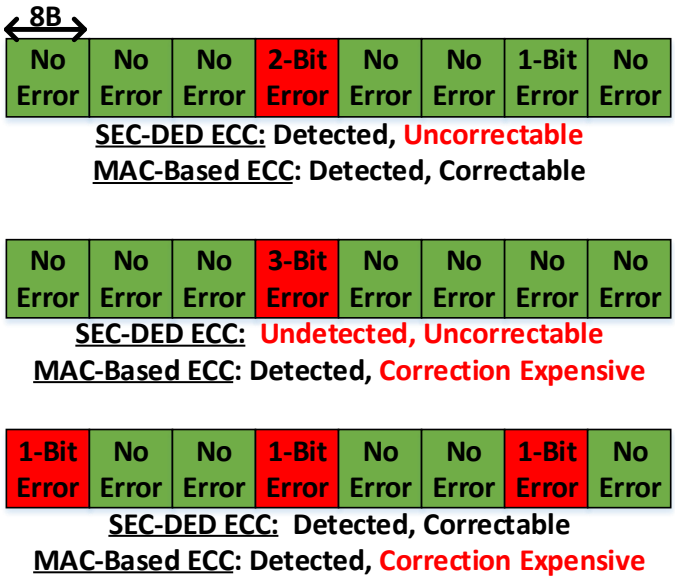


Figure 4.2 SEC-DED ECC vs MAC-Based ECC on a 64-byte Block. The examples show that the error correction capability of the two schemes is highly dependent on the number and position of bit-flips.

The most straightforward way to achieve MAC-based error correction without compromising security is performing a **brute-force flip-and-check** on each of the bits. When an integrity check fails, we attempt to correct the bit error(s) by flipping each bit in the memory block one by one and re-checking the MAC value.

Cost of Error Correction. A simple flip-and-check algorithm over 64-byte (512-bit) memory blocks will require a maximum of 512 flip-and-checks to correct single-bit errors, whereas correcting double-bit errors will require a maximum of 130,816 flip-and-checks (512 combination 2). Since state-of-the-art MAC algorithms, which are essentially composed Galois field multiplications, can be computed within a single cycle in hardware [121, 36], performing double error correction through brute-force attempts would be feasible.

Performance Implications. Even if we are increasing the cost of error recovery when employing a brute-force ECC scheme, this will have minimal impact on performance as

DRAM errors are rare occurrences. Fault analysis on Facebook's entire fleet of servers reveals that, on average, only 2.08% of the servers experience correctable (single-bit) errors per month, while just 0.03% of the servers experience uncorrectable (double-bit) errors per month [61]. Furthermore, the same study has shown that the majority of the servers affected by DRAM errors have at most 9 correctable errors per month.

Comparison with Standard Error Correction Standard ECC memory is able to correct single-bit errors per 8-byte words. For a full 64-byte block, we have up to 8-bit error correction – provided that we do not have more than a single-bit error per 8-byte word. On the other hand, the level of error correction provided by the brute-force approach we propose has the following characteristics:

1. The level of error correction provided depends on the amount of worst-case latency we are willing to tolerate. Correcting anything beyond 2 bit-flips inside a 64-byte block will require millions of cycles in the worst case.

2. Unlike Hamming codes, we cannot provide error correction at the granularity of 8-byte words. As a bit flip in one word will affect all the bits in the MAC tag, we can only perform checks over the entire 64-byte memory.

3. Standard error correction outperforms the flip-and-check scheme in the event where we have multiple single bit-flips spread across multiple 8-byte words. However, the flip-and-check approach can correct double bit-errors with reasonable overhead even when they occur within a single 8-byte word.

Table 4.1 summarizes the comparisons between MAC-based ECC and traditional ECC. As can be seen, MAC-based ECC provides a better error detection mechanism while enabling tamper-proof storage. On the other hand, the effectiveness of MAC-based error correction, as compared to traditional ECC depends on the location of the bit-flips. Figure 4.2 gives examples of different bit-flips and how SEC-DED and MAC-based ECC perform under those conditions.

4.5 Attacks Versus Hardware Faults

When merging error correction and integrity checking, we need to ensure that an attacker is not able to masquerade an attack as a hardware fault. The property of secure MACs make it impractical for an attacker to flip bits and pass them through the MAC-based error correction mechanism undetected.

If the attacker flips even a single bit in the cipher text, they will need to re-compute the full MAC to avoid detection – which cannot be done without knowing the protected secret key. Otherwise, all single bit-flips (whether in the cipher text or MAC) will be reverted by the error correction mechanism. Similarly, double bit-flips in the cipher text will also be reverted.

On the other hand, if an attacker flips more than two bits, an exception can be raised (similar to double bit-flips in standard ECC), or a more expensive correction can be attempted. As more than 2 bit-flips in a single memory block is a rare occurrence under normal conditions [61], the application can treat them as a potential attack and shut itself down. It should be noted that on standard ECC, three or more bit-flips within an 8-byte word cannot even be detected.

4.6 Evaluation

As discussed so far, merging the error correction and integrity checking mechanisms will significantly reduce the storage overheads. On top of this benefit, the MAC-based ECC scheme enables us to read the MAC in parallel with the data, reducing the amount of memory transactions required to verify the integrity of blocks. This can potentially result in improved application performance. In this section, we analyze the performance benefits resulting from this scheme.

CPU	3.2GHz, OoO, 4-cores, L1: 32KB, 8-way, L2: 256KB, 8-way, L3: 10MB, 16-way, shared
DRAM	4 channels, DDR3-1600
Memory Encryption	32KB, 8-way counter/MAC cache, 5-level Off-Chip Integrity Tree, (protecting 512MB region)
Benchmark	PARSEC 2.1, sim-med Input 4-thread parallelism

Table 4.2 Experimental Setup. CPU was simulated using MARSSx86, integrated with DRAMSim2 DRAM simulator.

4.6.1 Experimental Setup

We simulated a quad-core system using the MARSSx86 cycle-accurate CPU simulator [66], integrated with the DRAMSim2 memory simulator [97]. The details of the simulated system is given in Table 4.2. We modified DRAMSim2 to include the full memory encryption logic.

Benchmarks We run the PARSEC 2.1 benchmark [18] with the *sim-med* input set [1]. To better stress the memory system, we run 4-threads concurrently. We were able to successfully run 11 of the 13 applications in the benchmark suite on our simulator while the other two failed as they used instructions that are not fully supported by the CPU simulator.

Memory Integrity Tree We allocated a total of 512MB memory for secure data storage (i.e., protected by an integrity tree). We assume 3KB of on-chip SRAM is available [81] to securely store the top-level nodes of the integrity tree (Section 2.3). With this setup, the off-chip *baseline* integrity tree will be 5 levels deep. We configured the memory encryption engine with a 32KB, 8-way counter and MAC cache in all the experiments.

[1] The PARSEC benchmark suite with the sim-med input enables us to run the full parallel region of interest (ROI) from start to end on a detailed, cycle accurate simulator.

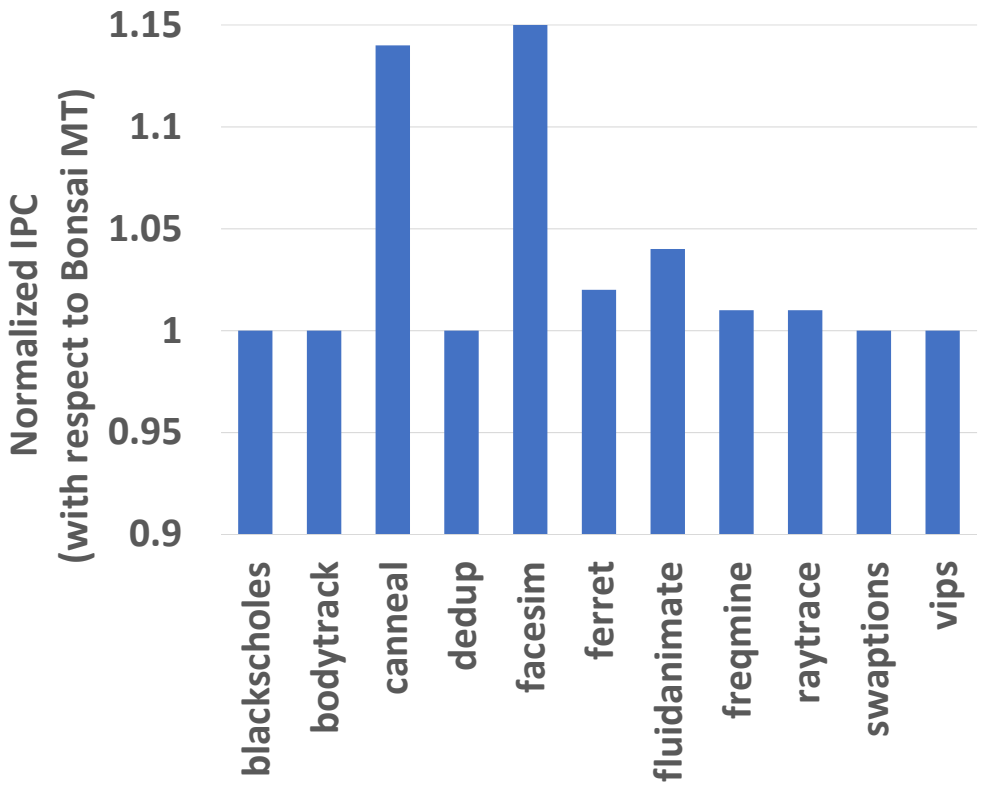


Figure 4.3 Application Performance with MAC-Based ECC. In addition to the significant storage savings, merging authentication and ECC results in modest speedups (compared to regular Bonsai Merkel Trees).

4.6.2 Results

Our simulation results show that, in addition to the significant space savings, MAC-based ECC improves IPC over the PARSEC benchmarks by an average of $\sim$ 3% (with up to $\sim$ 15% improvements) compared to Bonsai Merkel Trees. In Figure 4.3 we show performance improvements over the the PARSEC benchmark. About half of the programs in the benchmark show nearly zero performance improvement as they already benefit from the 32KB encryption engine cache and/or have low L3 miss rates.

Chapter 5

Reducing Counter Storage Overhead Using Delta Encoding

Authenticated memory encryption needs to store a message authentication code (MAC) and a counter value for each protected memory block. The MAC-based ECC scheme presented in the previous chapter reduces the overheads associated with fetching and processing MACs.

Even after optimizing the MAC storage, the counter storage and processing imposes a non-trivial overhead. Intel SGX, for example, stores a 56-bit counter for each encrypted 64-byte block, resulting in an $\sim$ 11% Furthermore, an integrity tree has to be constructed over these counters to prevent them from being tampered. In this chapter, we propose techniques for storing the per-block counters in a more compact manner – without impacting system performance.

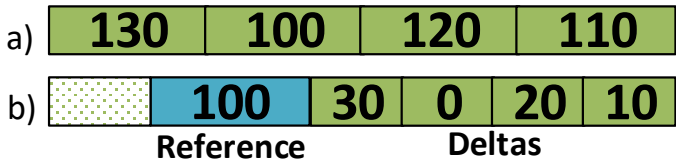


Figure 5.1 Frame-of-Reference Delta Encoding. We can reduce the space required to store an array of numbers by encoding the numbers as a delta to a single reference value. The array of integers in **(a)** above is delta-encoded using 100 as a reference. In this example, we can represent the delta values using just 5 bits per value, whereas storing the full integers would have required 8 bits per value. The storage savings will be much bigger when we apply this technique to large 56-bit counters.

5.1 Delta Encoding

Delta encoding is a data representation scheme that stores the difference (deltas) between two values, instead of storing the full values themselves. It is widely used for reducing entropy in a dataset before compressing it, and has also been proposed for use as a cache compression algorithm [68].

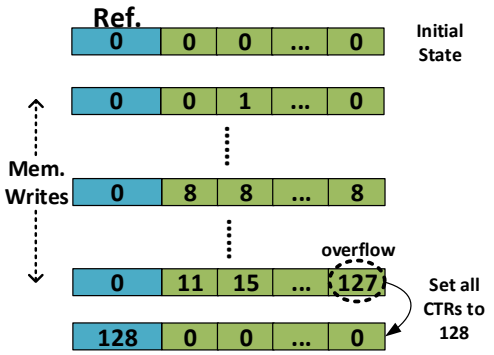
Delta encoding results in a more compact representation when the range of values appearing in the data is relatively small. Figure 5.1 illustrates a flavor of delta encoding, known as *frame-of-reference delta encoding*, that we will employ for compacting counter storage. Under this scheme, we store a single **reference value** and each counter is stored as a delta to the reference value. The array [130, 100, 120, 110] in Figure 5.1a, for example, is stored as deltas with respect to the reference integer 100: [130-100, 100-100, 120-100, 110-100] = [30, 0, 20, 10]. Note that we can represent the delta values using just 5 bits per value, whereas storing the full integers would have required 8 bits per value. The storage savings will be much bigger when we encode large 56-bit counter values, as we will discuss below.

To extract the original values, we "decode" the array by simply adding the reference value to each delta. This decoding operation will require minimal extra hardware (Section 5.5.1).

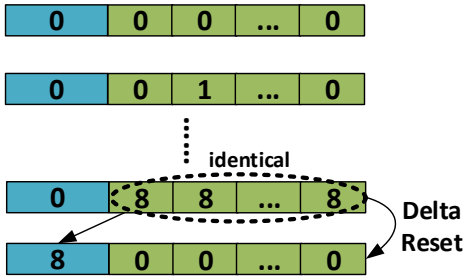
To avoid storing negative numbers, the reference value is typically picked to be the minimum value in the array. However, as it will become apparent in the following sections, we do *not* need to search for the minimum value in the array during the encoding process.

Delta Encoding of Counters. To delta-encode counters, we group multiple memory blocks into a **block-group**. Memory blocks that are part of the same block-group are encoded using a common reference value. For example, if we group 64 memory blocks to form a 4KB block-group, we will store a single reference value and 64 delta values.

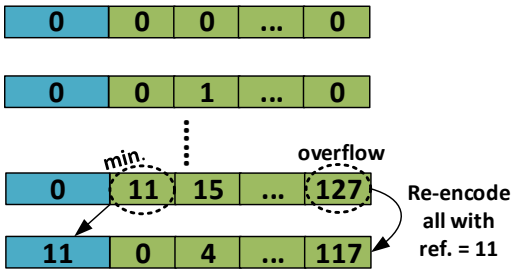
Delta encoding significantly reduces counter storage requirements. A more compact representation is possible as the delta values are much smaller than the full counter value. In our design, the reference value is 56-bits, similar to the size of a counter in



(a) Handling Delta Overflows: When a delta overflows, we re-encrypt the entire block-group using the largest counter value in the group (128 in this example). After that, we reset the deltas to zero and store the counter as the reference.



(b) Resetting Deltas: To reduce counter growth rate, we reset deltas to zero when they all converge to the same identical value. After resetting the deltas, we also increment the reference value.



(c) Re-Encoding Counters: When an overflow occurs, we will first attempt to re-encode the counters using smaller deltas. We simply subtract the minimum delta value (Δ_{min}) from all the deltas and increment the reference by Δ_{min}. If $\Delta_{min} == 0$, re-encoding is not possible.

Figure 5.2 Delta Encoding Counters. We apply the frame-of-reference delta encoding scheme to compactly represent counters. Delta overflows are handled by re-encrypting the block-group using the largest counter the group (shown in a). We apply two optimizations to reduce the rate of re-encryption (shown in b, and c)

Intel SGX, and would never overflow during the lifetime of a machine. On the other hand, our experimental results over the PARSEC benchmark show that even a 7-bit delta values are practical for most workloads (Section 5.5.1).

The encoding scheme we are proposing here provides the same amount of storage savings as split counters [121]. However, as we will detail in this chapter, delta encoding enables certain optimizations that could not be applied on split counters.

5.2 Counter Updates and Re-Encryptions

When encrypted DRAM blocks are initialized, all counters are initialized to zero. These values are represented with reference = 0 and delta = 0 (Figure 5.2). When a memory block is updated, the delta value for that specific block is incremented.

Delta Overflow & Re-encryption. Since the deltas are small integers, they can overflow after hundreds of memory writes. When a delta overflows, we need to re-encrypt the memory blocks in block-group with a new counter (Figure 5.2a). As the counter that just overflowed is the largest counter in the group, we will use it for re-encrypting the block-group. In addition, we increment the reference value by this overflowing counter's value, and set all the deltas to zero.

The re-encryption operation essentially involves sequentially reading memory blocks in a block-group and encrypting them using an identical counter value, i.e., the largest counter value in the block-group. Note that we do **not** need to re-encrypt the specific memory block whose counter just overflowed (and triggered the re-encryption operation).

Block-Group and Delta Sizes. The decryption pipeline will perform better if both the reference value and the associated deltas are stored in the same memory block, as both of these values can be loaded with a single read operation. There are multiple block-group and delta size combinations that would satisfy this criteria.

To test the effectiveness of our algorithms under low storage overheads, we evaluate our system using 7-bit deltas (in line with the 7-bit minor counters evaluated in [121]). With 7-bit deltas, we can fit a 56-bit reference counter and 64 delta values. Hence, we picked a 4KB (64 blocks) block-group. This reduces the storage requirement by a factor of 7 compared to storing the full 56-bit counters.

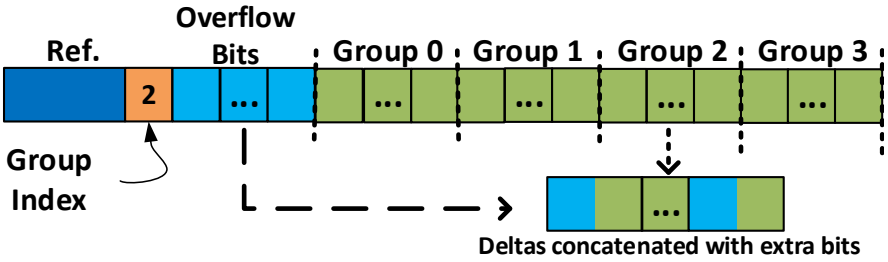


Figure 5.3 Dual Length Delta Encoding. We encode deltas using a dual-length encoding scheme, where a delta is supplied with extra "overflow bits" upon overflow. This scheme sacrifices optimality for low-overhead decoding .

5.3 Minimizing Overflow

The major limitation with delta encoding (as presented so far) is that small delta counters can overflow frequently – similar to minor counters in the split counters scheme [121]. Delta encoding, however, provides us with opportunities for reducing the rate of overflows. We present three techniques below for reducing the rate of delta overflow. In the next section, we will detail how these optimizations are triggered and handled by the hardware.

Dual-Length Encoding. The delta of more frequently updated blocks will overflow faster than the less frequently updated ones. Hence, instead of assigning 7-bit deltas to all blocks in a block-groups, it would be beneficial to assign more bits to deltas that are growing fast, and less bits to deltas that growing slow. Variable-length encoding is a mechanism that is used to store integers efficiently by assigning fewer bits to smaller integers.

However, decoding an array of arbitrary length integers would require numerous cycles – seriously impacting memory access latency. To mitigate this issue, we designed a constrained form of variable-length encoding – which we call dual-length encoding.

Figure 5.3 illustrates our dual-length encoding scheme. We group deltas in a block-group into 4 logical delta-groups (each containing 16 deltas). Each delta in this case is just 6 bits (instead of 7 bits as described above). With these slightly shorter deltas, we will have 72 unused bits in each block-group. These extra, unused bits are later used to

expand the delta-group that contains an overflowing value that cannot be represented using 6 bits. Each delta is expanded with an additional 4 bits upon overflow.

In the example in Figure 5.3, at least one delta in group 2 is overflowing. To avoid re-encryption, we assign the reserved overflow bits to group 2. We also set the group index bits to indicate the overflow bits are assigned to group 2. If group 2 or any other group overflows after this point, we will re-encrypt the block-group.

This constrained form of variable length encoding does not provide optimal storage savings, but significantly minimizes the decoding latency. The decoding operation also requires minimal extra hardware (Section 5.5.1).

Since this encoding cannot fully eliminate the possibility of overflows, we try to minimize overflows and re-encryption by employing the techniques discussed below.

Resetting Deltas. Memory accesses of real workloads commonly exhibit spatial locality. When memory writes have spatial locality, we can expect delta values of contiguous memory blocks to grow at a comparable rate. This phenomenon can be exploited to reset delta values when they converge to an identical value.

Consider the example shown in Figure 5.2b. Initially, the delta values are not all identical. Over subsequent updates, all the deltas converge to an identical value, i.e., 8. When this happens, we can simply reset the deltas to zero and update the reference value. The final state in Figure 5.2b shows how the reference and delta are reset. The reference value is incremented by 8, and all the deltas are set to 0.

Resetting the deltas in this manner reduces the rate of overflow. This mechanism is especially effective for workloads that have writes to sequential memory blocks.

Re-Encoding Counters. If a delta cannot be reset and overflows, we will attempt to avoid (or at least defer) re-encryption by re-encoding the counters using a larger reference value.

In Figure 5.2c, the last delta would overflow if we increment it on the next write (assuming 7-bit deltas). Instead, we re-encode counters with a larger reference, using the following algorithm:

1. Find the minimum delta, Δ_{min}, in the block-group ($\Delta_{min} = 11$ in Figure 5.2c)

2. Subtract Δ_{min} from all the deltas in the block-group.

3. Add Δ_{min} to the reference value.

In short, when a delta value overflows, we attempt to re-encode block-group's counters using smaller deltas. The re-encoding algorithm is effective only if all the deltas in the block-group are greater zero , i.e. $\Delta_{min} > 0$. If one or more deltas in the block-group are zero, this algorithm will not be able to reduce the delta values and the block-group will be re-encrypted.

The results we present in Section 5.5.1 indicate that this re-encoding scheme does *not* reduce the overall rate of re-encryption on most applications. However, on certain applications, it results in significant reductions in re-encryption rate – suggesting that the re-encoding algorithm will still be very valuable under certain workloads.

Lifetime of Non-Volatile Main Memories (NVMMs). Encrypting data in an NVMM can result in faster storage media wear out [127]. Frequent re-encryption of memory blocks that result from overflowing counters will exacerbate this problem. As the results we present at the end of this chapter illustrate, the delta encoding scheme we present in this work reduces the rate of re-encryption compared to other compact counter storage schemes [121]. This reduced re-encryption rate mitigates accelerated storage media wear out that can result from compact counter storage schemes.

5.4 Putting it All Together: Delta Encoding Implementation

We now consolidate the optimizations presented and discuss hardware implementation issues. Figure 5.4 shows the main components for implementing delta encoding/decoding and handling overflows.

The following extra hardware are introduced:

- **Decode Unit**: A small piece of hardware for extracting a delta value and adding it to the reference value is added. This component decodes counters as they are

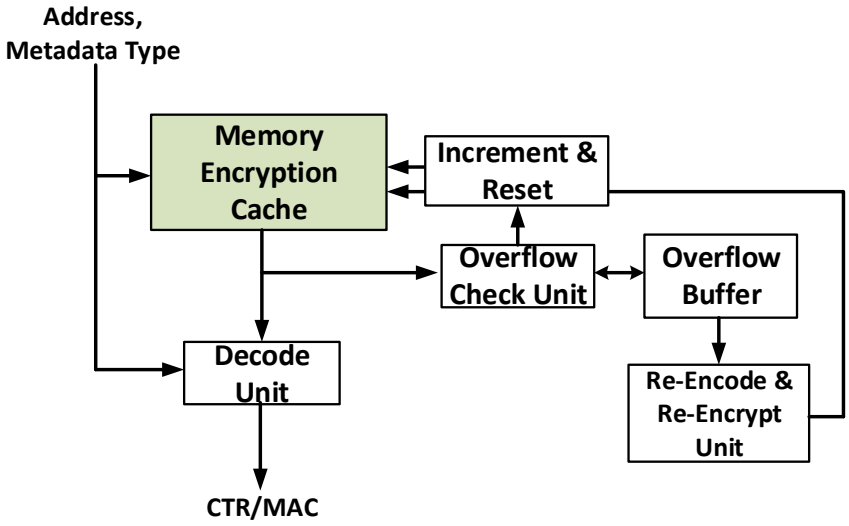


Figure 5.4 Implementation of Delta Encoding

being read out of the counter cache into the memory encryption engine. If the metadata that is being read out is not a counter, the decode unit simply selects the desired bits without performing any addition. The decoding logic is very lightweight, involving a bit extraction and an add operation. These operations can be completed in just 2 cycles at high clock frequencies (Section 5.5.1).

- **Increment and Reset Unit:** On a write operation, the counter value is updated by incrementing its delta value. This is done by reading out a line from the counter cache and incrementing the specific delta value. Before incrementing the delta, an overflow is checked. Finally, after a successful increment operation, the delta reset logic checks if all the deltas are identical. Since the increment and reset are done on a write, the extra cycles required for these operations will not impact application performance.

- **Re-encoding and Re-encryption Unit:** When an overflow is detected by the overflow check unit, the address and counters of the block-group that needs to be re-encrypted is added to the re-encryption queue. The re-encryption engine will read addresses from this queue and re-encrypt block-groups in turn. Before

attempting an expensive re-encryption operation, however, the re-encryption engine will attempt to re-encode the counters with smaller deltas using the algorithm described earlier. This queue does not need to be large as the rate of re-encryption will be low (Section 5.5.1).

Current industrial memory encryption implementations already contain hardware that can potentially be leveraged for re-encrypting block-groups. Intel SGX has logic for swapping out pages from a secure memory region (known as the enclave page cache) and writing it to an operating system accessible region [81]. Swapping out secure pages involves a re-encryption operation akin to the one we need to perform on overflows. Similarly, AMD's memory encryption hardware includes an ARM Cortex-A5 microcontroller fitted with cryptographic accelerators, and is responsible for key management [50]. Both of these existing hardware can be enhanced to implement the re-encryption operation without introducing major hardware components.

5.5 Evaluation

In this section, we will evaluate the performance impact and rate of block-group re-encryption. We will also evaluate the performance benefits of combining delta encoding with the MAC-based ECC technique from the previous chapter.

Experimental Setup: We use the same set of benchmarks and full-system simulation setup from the previous chapter (see Section 4.6).

5.5.1 Results

Counter Decoding Overhead. Counters need to be decoded by concatenating the deltas with extra overflow bits (or zeros), and then summing the base and delta. To measure the decoding overhead, we synthesized the decoder logic to IBM's 45nm silicon-on-insulator (SOI) technology library using Synopsis Design Compiler. With this

Program	7-bit Minor Split CTR [121]	7-bit Delta	Dual Length Delta
facesim	880	113	176
dedup	725	51	14
canneal	167	167	128
vips	77	77	24
ferret	33	23	5
fluidanimate	4	4	0
freqmine	3	0	0
raytrace	2	2	0
swaptions	0	0	0
blackscholes	0	0	0
bodytrack	0	0	0

Table 5.1 Average Number of Re-Encryptions per *1 billion cycles*. Average across three full executions to account for variations in multi-threaded execution.

setup, the decoding logic is able to complete within 2 cycles for frequencies up to 4GHz, and has negligible area overhead ($\sim 0.002mm^2$). Our simulations account for these 2 extra cycles. Since we used an older technology node, our estimates will be slightly pessimistic compared to what can be achieved using newer silicon technology.

The other operations associated with delta encoding (counter reset, re-encoding, and re-encryption) are triggered during a write, and hence will not directly impact read latency. Furthermore, current industrial memory encryption implementations already contain hardware that can be leveraged for re-encoding and re-encryption.

Re-Encryption. If a delta overflows, re-encryption is triggered on the block-group (Section 5.2). Table 5.1 compares the re-encryption rate for different counter representations. It can easily be seen that, for memory intensive workloads, delta encoding (combined with our delta reset algorithms) significantly reduces the number of re-encryptions compared to split counters [121]. As discussed above, this makes delta encoding more efficient and non-volatile memory friendly. Dual-length deltas perform better than 7-bit deltas overall. On facesim, however, dual-length deltas have increased re-encryption rate as multiple delta-groups overflow the default, shorter 6-bit

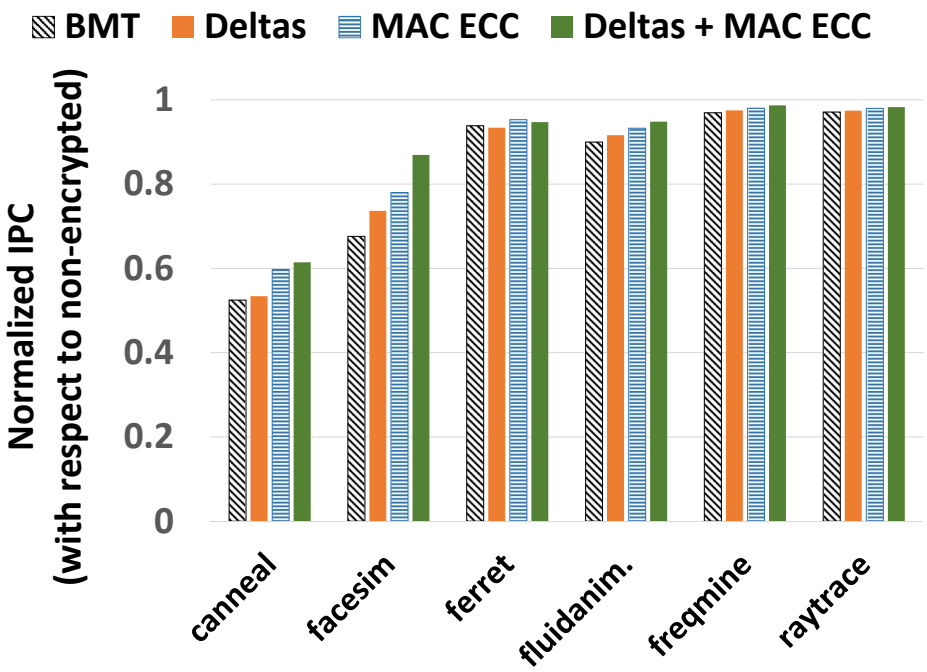


Figure 5.5 Performance Impact of Authenticated Memory Encryption. The proposed storage optimizations (Delta Encoding and MAC over ECC Memory) reduce the performance impact of memory encryption compared to Bonsai-Merkel trees (BMTs). The two techniques can be combined together get maximum benefits.

delta storages concurrently, and cannot be all extended using the reserved overflow bits.

Overall Application Performance. Figure 5.5 presents the performance impact of authenticated memory encryption when the two storage optimizations proposed in this thesis are enabled. Authenticated encryption has no measurable impact on some of the applications (bodytrack, vips, blackscholes, swaptions). These applications do not benefit from any further optimizations and are not shown in the figure. Our simulation results show that enabling delta encoding only results in modest application performance improvements. Delta encoding is able to improve application performance

for two reasons. First, compact counter storage reduces the depth of the Bonsai Merkel trees – resulting in fewer extra memory reads. For the system we evaluate in particular, the depth of the Bonsai Merkel tree is reduced from 5 levels to 4 when counters are delta encoded. In addition to reducing tree depth, the counter cache hit-rate will also improve as we are fitting more counters into a single memory block. Our simulation models do not include the separate re-encryption logic, but its performance impact will be minimal as re-encryption can be performed without suspending the rest of the system.

Combining Delta encoding with MAC-based ECC. Our simulation results show the storage-optimized tree reduces the performance impact of memory encryption (due to fewer off-chip reads) by an average of 5% over the PARSEC benchmark suite (with a maximum of 28% improvement). Authenticated encryption has no measurable impact on some of the applications (bodytrack, vips, blackscholes, swaptions). These applications do not benefit from our optimizations.

Part II: A Lightweight Hardware Extension for Enforcing Operating System Integrity

Chapter 6

Operating System Kernel Security

The security and trustworthiness of applications depend on protections provided by the operating system. A malware that is able to run at the same privilege level as the OS kernel can gain full control of a system and even render malicious activities invisible to the rest of the system. As a result, software defenses or forensics techniques cannot reliably operate on top of a compromised OS. Numerous protections are continuously being introduced to harden operating systems against attacks. These mitigations have made it significantly harder to compromise modern operating systems.

This chapter provides an overview of the attack vectors exploited by kernel-mode malware and rootkits, and describes existing kernel protections and their limitations.

6.1 Kernel-Mode Execution and Rootkits

The most severe compromises on an operating system would result in running arbitrary malicious code in kernel-mode (i.e., code that runs with kernel privileges). Some types of kernel-level malware, referred to as rootkits [1], can even hide their existence and remain undetectable on the system [41].

Despite the increasing sophistication and volume of attacks on the operating systems, kernel-mode malware and rootkits need to fundamentally rely on one of these three techniques to achieve privileged execution:

[1] The term rootkit is sometimes used to describe non-malicious kernel or root code – such as a security monitor. In this dissertation, we use the word rootkit to refer to malicious and stealthy kernel-mode malware.

- **Installing Malicious Drivers:** This is one of the oldest and most straightforward ways to run a malicious privileged code. Drivers/kernel modules can potentially tamper with any data in the system since they run at the same privilege level as the kernel. Modern operating systems make malicious driver installations difficult by requiring drivers to be signed by a trusted key [9, 98].

- **Privilege Escalation Attacks:** Attackers have repeatedly managed to discover mechanisms to leak or overwrite kernel memory. These mechanisms range from exploiting memory safety and integer overflow/underflow errors in kernel-mode code [84, 91, 19], to exploiting race conditions [87, 101, 116], to abusing hardware bugs [100]. Such exploits are convenient for injecting code into kernel memory or for executing user space code at an escalated privilege [26]. In addition to exploits that function by corrupting or leaking kernel memory, it has been shown that nearly all major operating systems had the same bug in their system call handlers that enabled directly running user space code in kernel-mode [83]. Evidently, there are numerous avenues for privilege escalation attacks.

  Privilege escalation attacks are not mitigated by driver signature checks as such attacks can inject code into kernel space without passing through the driver loader logic. Furthermore, privilege escalation bugs which allow overwriting portions of kernel memory can themselves be used to disable code signature checks [19].

  Operating systems are too large and complex to completely eliminate vulnerabilities that can lead to privilege escalation attacks. As we will describe in the next section, modern CPUs enable operating systems to restrict which code pages can be executed in privileged mode. This hardware feature makes privilege escalation attacks more challenging even in the presence of certain vulnerabilities.

- **Code Patching:** It is also possible for attackers to modify kernel code to redirect execution or bypass certain security checks [41]. Since operating systems typically mark their code as read-only once loaded into memory, such type of code patching attacks need to rely on a bypass that enables writes to protected code pages or files.

6.1.1 Rootkit Stealth Techniques

Once attackers introduce malicious code into the system, they can employ additional stealth techniques to hide their existence. This is achieved by making network activities, processes, and files invisible to user space programs. Techniques for achieving stealth can be grouped into three categories [41]:

- **System call and interrupt descriptor table hooking:** On a system call, the kernel consults the system call table (also referred to as System Service Descriptor Table in Windows) to locate the kernel function that corresponds to a system call. On some CPU architectures, both interrupts and system calls are handled using a single dispatcher code. On the other hand, on x86 CPUs, the kernel and the CPU rely on the interrupt descriptor table (IDT) to execute the kernel code that corresponds to a specific interrupt request. System call tables and IDTs are essentially a list of function addresses. Rootkits can overwrite entries in these tables to execute malicious code on a system call or interrupt [41]. By hijacking system calls and interrupts, attackers can control what values the kernel returns to user space programs. For example, a rootkit can hide a file or process by removing it from a list that a system call returns to a user space program.

  CPU architectures typically have a dedicated register that stores addresses of kernel entry points for system call and interrupt handling (e.g., on RISC-V, the stvec register typically holds the address of the kernel's system call/interrupt handler, while the IDTR register holds the address of the IDT in x86). Since direct modification of system call tables or the IDT might trigger certain rootkit detectors, some rootkits create a new IDT or system call handler and overwrite the address in these special purpose registers to point to the new addresses [41].

- **Hooking code pointers:** It is relatively easier to detect changes to pointers in the system call or interrupt descriptor tables. As a result, advanced rootkits try to redirect execution by hijacking dynamic code pointers in the kernel. Such type of stealthy hooking is preferred by today's advanced rootkits [123] as detecting malicious changes to an arbitrary code pointer is a more challenging task.

- **Direct Kernel Object Manipulation (DKOM):** A kernel typically has various well-documented data structures to track system information such as running processes/threads and list of open network connections. Rootkits can manipulate these data structures directly to make their execution and network footprint invisible [41].

6.2 Shielding the Operating System from Attacks

Due to the prevalence and severity of attacks that target operating systems, numerous techniques for protecting the kernel have been extensively studied and deployed. In this section, we will highlight these protections, and motivate the necessity of the hardware extensions we propose in the next chapter.

6.2.1 Secure Boot

Running a trustworthy operating system requires that the system initially boots into an untampered kernel. If the BIOS loads a compromised kernel, or if the kernel loads a compromised driver, then all other protections built into the operating system can be rendered ineffective.

Secure boot refers to a mechanism that verifies the integrity of the BIOS and kernel to ensure the system boots into an untampered state. This is achieved by deploying a digitally signed or encrypted kernel and BIOS code, which the bootloader can later verify using the appropriate encryption key. It is aimed at detecting modifications made prior to booting, but provides no guarantees the operating system will remain untampered once it starts running.

The protections we discuss below, and the protections we propose in the next chapter, generally assume that a secure boot mechanism ensures the system initially boots into an untampered kernel. That being said, secure boot implementations are certainly *not* impenetrable and can be vulnerable to certain advanced attacks [20].

6.2.2 Driver Signing

Operating systems rely on driver signing to prevent malicious drivers from being loaded into the system. Recent versions of Microsoft Windows and macOS only allow signed drivers by default [126, 9]. The Linux kernel also supports kernel module signature verification.

Even if the widespread adoption of driver signing has made it significantly harder to install malicious drivers, attackers have repeatedly found ways to bypass this protection. Attacks that disable code signing by exploiting a bug in the kernel or a legitimate driver have been discovered in the wild [31, 19]

Furthermore, driver signing mechanisms cannot defend against privilege escalation attacks (discussed above) since those attacks avoid going through the driver loading mechanism altogether and do not trigger any driver signature checks.

6.2.3 Supervisor Mode Access/Execution Prevention

In a typical virtual memory system, a bit in the page table entries (PTEs) marks a page as user-mode or kernel-mode page. The CPU can use this bit to prevent user space programs from accessing kernel memory.

Modern CPUs provide another layer of page-based protection to prevent arbitrary user space code from executing with escalated privileges. These CPUs check the user/supervisor bit in the PTEs, and prevent code from a user-mode page from being executed in kernel-mode. Intel's Supervisor Mode Execution Prevention (SMEP) [82] and ARM's Privileged-Execute-Never (PXN) [75] are two examples of this technology.

Intel Supervisor Mode Access Prevention [82], another related protection, uses the user/supervisor bit in PTEs to prevent the kernel from directly accessing user space memory. This prevents the operating system from erroneously dereferencing a pointer to a user space memory (which could potentially contain malicious instructions or data set up by a user space program).

These CPU extensions make privilege escalation attacks quite challenging even in the presence of kernel bugs. These protections, however, are under the full control of the

operating system itself. As a result, an oversight or bug in the kernel could be exploited to disable these protections. Kemerlis et al., for example, have demonstrated how the fact that the Linux kernel "mirrors" the entire physical memory in its own address space can be abused to bypass SMEP and PXN [51]. Memory safety bugs have also been exploited to circumvent PXN and tamper with kernel data structures on Android [120].

More importantly, SMAP and SMEP can be disabled by clearing a single bit in x86's CR4 control register. As a result, code-reuse attacks can be used to disable SMAP/SMEP by modifying the value of the CR4 register [58, 104]. The additional kernel integrity monitoring layers in Windows (discussed below) attempt to defend against this bypass technique by monitoring the state of the CR4 register [126].

6.2.4 Kernel Integrity Monitoring and Enforcement

Eliminating all potential vulnerabilities from an operating system remains to be an elusive goal. Therefore, to safeguard the kernel's integrity even in the presence of vulnerabilities, certain operating systems (notably Microsoft Windows and some Android distributions) provide an additional security layer for monitoring/enforcing kernel integrity [12, 126].

64-bit versions of Microsoft Windows are protected by PatchGuard – a Kernel Patch Protection[2] mechanism that is directly integrated into the kernel. PatchGuard periodically scans kernel code, CPU control registers, and data structures that are commonly targeted by malware and rootkits (e.g., interrupt descriptor tables, system service descriptor tables, the kernel stack, and kernel configuration flags) [6, 126].

Unfortunately, kernel integrity protections that run at the same privilege level as the kernel can be disabled by exploiting a vulnerability in the kernel itself. As a result, PatchGuard has been subject to attacks that relied on kernel exploits [6]. More secure implementations run the integrity protection module in a hardware-isolated environment – such as a separate virtual machine or a TrustZone secure world (discussed

[2]Part of the original motivation for Windows' kernel patch protection feature was improving system stability by preventing unsafe changes to the kernel even by non-malicious third-party programs, such device driver and anti-virus software.

below). Windows 10 and Android distributions used by major vendors are examples of commercial operating systems that have adopted this approach.

Virtualization-Based Kernel Protection. Virtualization-based kernel security solutions deploy the kernel in a virtual machine (VM) running on top of a hypervisor. Since the hypervisor has full control over VMs running on top of it, powerful defenses can be implemented inside the hypervisor or inside a separate VM. Defenses implemented in this manner have an advantage over defenses built directly into the kernel as the hypervisor and the different VMs are fully isolated by the hardware – making them harder to attack.

Previous works have demonstrated how hypervisors can be used to securely monitor activities in the guest OS [32, 47, 67], to detect code pointer hooking in the kernel [114], and prevent malicious privileged code execution [102, 94].

Windows 10 has also recently introduced a virtualization-based kernel protection scheme [126]. This protection enables security-critical tasks such as driver integrity verification, kernel integrity enforcement, and credential management to run in a separate virtual machine that is fully isolated from the Windows operating system. This isolation protects security-critical processes from being compromised by a privileged malicious code.

One challenge of virtualization-based security is that the hypervisor itself has a non-trivial attack surface [12]. For instance, multiple remote code execution and privilege escalation vulnerabilities have been discovered in Microsoft Hyper-V in 2018 alone [89, 88, 86]. To counter this issue, additional protections for defending the hypervisor itself have been proposed [113, 112, 13]. A tiny hypervisor that is amenable to manual audits and verification has also been presented in [102]. While these are all promising approaches, creating a fully verified commercial hypervisor that is bug-free remains to be a hard problem.

Aside from the security issues, running an OS in a virtualized environment also incurs performance overheads of up to 30%, which is undesirable for certain deployments [53]. Furthermore, a study on the Microsoft Windows' virtualization-based protection has shown that system calls from untrusted applications can be up to 200x (2000%)

slower as a result of page table entry modifications the hypervisor needs to make on each system call [118].

TrustZone-Isolated Protection. Some ARM CPUs implement TrustZone, a hardware extension that enables isolating software and hardware components into secure and non-secure worlds [73]. The hardware prevents the non-secure world from accessing the resources of the secure world.

Ge et al. proposed using this hardware feature for protecting kernel integrity monitors [34]. Android distributions deployed by major vendors use this approach to isolate the kernel integrity enforcement mechanisms from the main kernel [12, 16]. They leverage TrustZone's capabilities by running the main operating system (i.e., Android) in the non-secure world, while running another lightweight (trusted) kernel in the secure world. Security-critical tasks such as credential management and integrity checking run on top of the trusted kernel, shielded from the main operating system.

Protections based on TrustZone, or similar trusted execution environment (TEE) are preferred over the virtualization-based approach as TEE-based protections do not need to rely on a large and complex hypervisor for isolation [12]. Nevertheless, TEEs cannot fully thwart attacks against a vulnerable code running in the hardware isolated secure-world, and attacks have been demonstrated against multiple generations of mobile TEEs [96, 16, 85, 15]. In fact, trusted kernels commercially deployed in TEEs still lag behind mainstream operating systems in terms of built-in exploit mitigations [16].

6.2.5 Hardware Mechanisms for Kernel Integrity

A number of previous works have proposed relying on hardware extensions or peripheral devices for monitoring the integrity of the kernel [52, 13, 62, 57, 54, 70].

One category of the hardware-level defenses involves using a hardware peripheral, typically attached to the PCI bus, to take a snapshot of the physical memory [52, 70, 13]. The snapshot is then analyzed to verify the kernel's integrity. The hardware peripheral could be a custom hardware [70], a GPU [52], or a network interface card [13]

Repeatedly taking full memory snapshots can be slow and expensive, especially on systems that have a large memory. Furthermore, transient memory modifications that happen in between snapshot intervals can be missed [62]. There is another class of protections that avoid periodically taking full memory snapshots, and instead monitor the memory bus [54, 57, 62]. However, these bus snooping defenses require a significant amount of new hardware, including a standalone microcontroller that has its own private memory, DRAM controller, and DMA engine. In addition, bus monitoring hardware that only snoop the DRAM bus require periodic cache flushes to ensure they see all changes to memory [57].

These hardware mechanisms are designed to be minimally invasive by operating independent of the CPU. However, it is challenging to identify the semantic meaning of raw bytes in memory or on the data bus without having access to additional information [46]. For example, reliably determining which memory locations constitute the page table entries or interrupt descriptor tables is quite challenging without looking at the CPU's control registers.

6.2.6 Limitations of Existing Defenses: A Summary

The protections discussed in this chapter make compromising the OS or planting a rootkit significantly more challenging. However, as we have highlighted above, these defenses still suffer from a number of limitations and implementation challenges. Table 6.1 summarizes these limitations and challenges.

One common theme that can be seen is that a vulnerability in the hypervisor, main kernel, or trusted (isolated) kernel can undo any of the software-based protections. And such types of system software are too large to be bug-free – a point which is evident from the series of attacks we have highlighted above.

The protection we present in this work is aimed at protecting the kernel's integrity even in the presence of driver or kernel vulnerabilities. As we will describe in the next section, the hardware extensions and software modifications required by our protection are minimal, thereby reducing the attack surface.

Protection	Known Limitations or Drawbacks
Secure Boot (§6.2.1)	Can be undermined by vulnerabilities in bootloader; provides no guarantees the operating system will remain untampered once it starts running.
Driver/Kernel Module Signing (§6.2.2)	Can be disabled by exploiting OS or driver vulnerability; cannot catch privilege escalation attacks that do not directly load drivers
Supervisor Mode Execution/Access Prevention and Privileged Execute Never (§6.2.3)	Run under the full control of the untrusted operating system
In-Kernel Integrity Monitoring (§6.2.4)	Runs at the same privilege level as the untrusted kernel
Hardware-Isolated Integrity Monitoring (§6.2.4)	Requires complex and high overhead software stacks (such as a hypervisor and/or a separate trusted kernel), which are still prone to exploits in the presence of software bugs.
Hardware-Level Kernel Memory Monitoring (§6.2.5)	Implementations may require significant amount of new hardware, periodic full memory scans, and periodic cache flushes. It is also challenging to identify the semantic meaning of raw bytes in RAM without having access to more information about the system's state.

Table 6.1 Existing OS Kernel Protections and their Limitations/Drawbacks.

Chapter 7

Neverland: A Lightweight Hardware Extension for Safeguarding Operating System Integrity

The previous chapter discussed how existing OS protections (implemented within a kernel, a hypervisor, or a trusted execution environment) have repeatedly been subjected to attacks as a result of their size and complexity. In this chapter, we propose a lightweight, hardware-based protection that enables operating system kernels to protect themselves against tampering – even in the presence of software vulnerabilities. The security approach we describe here, dubbed Neverland, does not rely on hypervisors or additional kernels running in trusted execution environments – thereby eliminating the complex extra components required by existing protections.

Neverland's protections work by stripping away the operating system's ability to change values of specific memory locations and configuration registers once the boot process is complete. Security-critical memory locations, such as security configuration flags, kernel code, and kernel read-only data, do not generally need to be modified once the system is done booting (there are exceptions to this, which we will discuss in Section 7.5). Our proposed hardware can be configured at boot time **to prevent the introduction of any new privileged code** into the system once the kernel is initialized, and **to disable all writes (at the hardware level) to kernel code and read-only data**. Since even the kernel is prevented by the hardware from modifying security-critical code, data, and configurations, software vulnerabilities in the kernel cannot be exploited to alter these memory locations.

We validated the efficacy of these protections by prototyping the hardware extensions in Spike (the official RISC-V emulator) and running a minimally modified version of the Linux kernel on the emulator. Our evaluations show that the hardware extensions required by Neverland do not incur any appreciable silicon or energy cost, and that integrating the proposed protections does not incur any performance overhead beyond the 10s - 100s of milliseconds that are required to setup permissions at boot time.

7.1 Threat Model

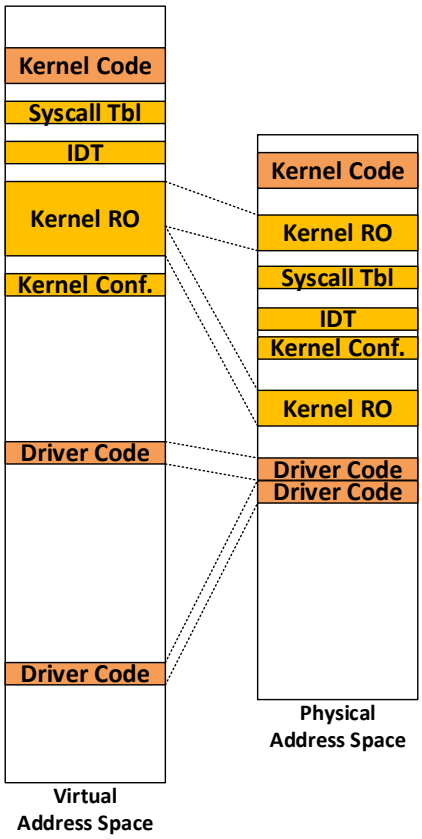
We consider a threat model that is similar to what is typically assumed by OS kernel protection mechanisms. We rely on the existence of a secure boot mechanism that ensures the system boots into an untampered kernel. However, we assume the OS kernel itself could have exploitable software vulnerabilities. The attacker's aim is to use these vulnerabilities to tamper with kernel code/data, to execute kernel-mode malware, or to hide malicious activity.

7.2 Design and Implementation Overview

In this section, we present the ingredients that makeup Neverland's protection mechanisms. We explain why these mechanisms reduce the operating system's attack surface in Section 7.6.

Write-Once Memory Regions. The kernel code and certain critical data structures (e.g., system call tables, security configuration flags) do not typically need to change once the system is initialized. Even the OS does not need to modify these data structures once they are initialized (there are exceptions to this claim, which we will discuss in Section 7.5).

Our hardware extensions enable the operating system to disable all *subsequent* writes to selected regions of memory that contain kernel code and security-critical static data –



(a) Memory Layout

Start Addr.	End Addr.	W	P	X	L	
0x249000	0x3f7400	0	1	1	1	(Kernel Code)
0x475530	0x475e50	0	1	0	1	(Kernel RO)
0x664050	0x664a70	0	1	0	1	(Kernel RO)
0x484320	0x484B20	0	1	0	1	(Syscall Tbl)
0x564840	0x565040	0	1	0	1	(IDT)
0x586340	0x586740	0	1	0	1	(Kernel Conf)
0x790300	0x790300	0	1	1	1	(Driver Code)
0x000000	0x000000	0	0	0	1	---

(b) Permission Table

Figure 7.1 Example Memory Layout and Physical Memory Permissions. Neverland enforces "write-once" and non-privileged-code-only restrictions by using a hardware table that stores permissions associated with protected physical address ranges. These permissions are enforced independent of the page permissions maintained by the OS. The permission table shown in (b) illustrates how the memory layout shown in (a) can be protected. For example, kernel code and read-only data are marked as read-only and locked. Once the locked bit for a permission entry is set to 1, even the operating system (or a malicious privileged code) cannot revert the permission entry until system restart. Hence, marking an initialized memory region as read-only + locked turns it into a "write-once" region. Note that virtually contiguous memory locations may not be necessarily be stored in a physically contiguous manner and as a result, we might need multiple permission entries for those regions.

regardless of page permissions or the software privilege level. Furthermore, once these memory regions are marked as read-only, they can be "locked" so that even the kernel cannot revert these permissions. Marking a memory region as locked + read-only once it is initialized effectively turns it into a "write-once" region.

Note that this write-once capability is entirely independent of the virtual memory system. As a result, code/data written in these locked regions will remain protected even if a vulnerability in the kernel is exploited to make read-only pages writable, or to make kernel pages accessible to user space programs. The only way these "locked" and read-only memory regions can be modified or freed is by rebooting the system. This protection approach obviates the need to continuously run an integrity checking thread or virtual machine, or the need to instrument the kernel and redirect page permission management.

Non-Privileged-Code-Only Memory Regions. A kernel-mode malware or rootkit will eventually need to execute its own code with escalated privileges. Neverland's second component is targeted at preventing the introduction of arbitrary privileged code into a system – even in the presence of vulnerabilities in the OS or drivers.

Neverland enables us to mark certain regions of the *physical* address space as containing kernel-mode code. **All other regions of memory *cannot* contain any kernel or driver code** (i.e., by default, memory regions are only accessible in user-mode). The CPU cannot fetch code from a physical memory location that is not marked as privileged while it is running in kernel-mode – and vice-versa. Similar to the "write-once" regions above, this invariant is enforced by a hardware extension that is independent of the virtual memory system – making it resilient to attacks even in the presence of operating system bugs.

This feature has two crucial distinctions compared to the Intel SMAP [82] and ARM PXN [75] features described in the previous chapter :

- SMAP and PXN rely on page permission bits and are under the total control of the operating system. As a result, a flaw in the operating system can be exploited to bypass these features [51, 58, 104]. On the other hand, the non-privileged-code-only memory region proposed here tracks permissions of physical

addresses (instead of relying the OS's virtual memory system) and relies on a hardware-managed table that is largely independent of the OS.

- Once memory locations are categorized as privileged-code and non-privileged-code-only regions (preferably immediately after the kernel finishes booting), the permissions can be locked from subsequent changes – even by the most privileged software in the system.

This feature, combined with write-once memory regions can be used to ensure all memory regions that contain privileged code cannot be patched, even by the OS or malicious drivers. Conversely, we can also ensure that writable memory regions can only contain non-privileged code. This configuration is illustrated in Figure 7.1. We can set kernel and driver code as "locked" since they do not typically need to change once loaded into memory. In short, flagging memory regions as locked or non-privileged-code-only makes it extremely hard to introduce new malicious privileged code into the system.

Locking Memory Permission Registers. The write-once and the non-privileged-code-only memory restrictions are specified using permission configuration registers (discussed below). The values of these configuration registers are intended to be set at boot time to minimize the attack surface.

After their values have been set, the ability to lock these permission registers against any modification would significantly improve the security of the system. If these configurations are not locked, vulnerabilities in the OS can be exploited to reset them – similar to attacks that disable SMAP/SMEP defenses by resetting control registers [58, 104].

Locking Configuration Status Registers. The CPU determines the location of interrupt descriptor tables (IDTs) and system call dispatchers using values stored in configuration registers. An attacker can trick the CPU into executing arbitrary code by overwriting the addresses stored in these registers (provided the attacker has already injected code to be executed into the kernel memory or has bypassed SMEP/PXN).

To provide an additional layer of security, existing kernel integrity mechanisms such as PatchGuard and HyperGuard [126] monitor configuration registers to detect malicious modifications. Instead of continuously monitoring changes to these registers in software, "hardware-locking" them from subsequent modification provides better security while eliminating the additional monitoring overhead. We discuss how register locking can be implemented in hardware in Section 7.3.

Permission Tables. The memory protections described above require a lightweight hardware to be integrated into the CPU. This additional hardware will maintain a small memory permission table that contains the physical address ranges and their associated permission bits. Permission bits specify whether a memory range is writable or "privileged-executable". A third permission bit is used to "lock" the permission entry (as discussed above). Figure 7.1b shows what the permission table's contents would look like for protection setup shown in Figure 7.1a.

A number of embedded CPU architectures support a form of memory permission tables. For example, the ARM Memory Protection Unit (MPU) and the RISC-V Physical Memory Protection (PMP) scheme are optional components that can be used to enforce permissions on physical addresses[115, 74]. They are a preferred mechanism for implementing a lightweight memory protection scheme in low-resource microcontrollers that do not have virtual memory support [11, 24]. These extensions, however, are not typically implemented by CPUs that have virtual memory support. Neverland uses a variation of such tables for enforcing memory permissions. We leverage immutable hardware tables and registers to provide low-overhead operating system protections.

We will dive into the details of the hardware implementation in the next section.

Permission Initialization and Enforcement. A kernel-mode code configures the permission tables immediately after all kernel code and read-only data are loaded into memory. If a memory region is covered by multiple overlapping permission entries, the most restrictive set of permissions are enforced. To prevent an attacker from specifying additional privileged code regions, unused permission entries must be locked as shown in the last row of Figure 7.1b. Initializing permissions immediately after secure boot,

and locking any unused permission entries takes away the attacker's ability to load arbitrary kernel-mode code.

What Memory Regions Are Protected? Neverland is ideal for protecting security-critical code and data that are loaded or set once (preferably at boot time), and do not need to change once the system is running. A typical configuration for protecting an OS would be composed of the following set of permissions, which are to be set immediately after the OS is done loading code and data into memory:

- **Core kernel code (text section):** privileged, read-only, executable, locked
- **Read-only kernel data (RO section):** privileged, read-only, locked
- **System call table:** privileged, read-only, locked
- **Interrupt descriptor table (on x86):** read-only, locked
- **Kernel Configuration (e.g., code signing enforcement configurations):** privileged, read-only, locked
- **Driver / loadable kernel module code:** privileged, read-only, executable, locked

Note that the code and data regions listed above are controlled or periodically scanned by a kernel integrity mechanism such as PatchGuard and HyperGuard [126]. The ability to irreversibly disable writes (until system restart) to these regions eliminates the need to monitor these regions using a continuously running software.

On the Linux kernel, the kernel code (text) and read-only data sections are typically stored in a physically contiguous region. So we will only need one permission entry for each. Even a 5-entry permission table would be sufficient to protect the critical components of the core kernel listed above.

On the other hand, loadable kernel modules/drivers are not loaded in a physically contiguous memory region, and as a result, protecting them would require dozens of entries in the permission table. This is undesirable as having a large hardware table incurs high overhead and cost. We will discuss how we tackle this challenge in Section 7.4.

Function Pointers. As discussed in the previous chapter, function pointers in the operating system could be overwritten by rootkits to hijack execution of system calls or driver code [123]. Function pointers that are typically hooked by rootkits rarely change their values once they are initialized [114]. Neverland could be used to lock such function pointers after they are initialized. Wang et al. have shown such function pointers can be relocated to dedicated read-only pages [114]. Their pointer relocation approach can be leveraged by Neverland to prevent malicious function pointer hooking. However, function pointers that need be updated multiple times while the system is running cannot be locked by Neverland.

7.3 Hardware Requirements

As highlighted above, the main additional component that is required is a table that stores and enforces the physical memory permissions. Each permission entry needs to store three pieces of information (Figure 7.1b):

- The physical address range (start and end address) for the memory region
- Privileged bit: if this bit is *not* set to 1, the CPU cannot read data or fetch code from this physical memory range while executing in privileged mode. **By default, all memory regions are non-privileged – regardless of page permissions.**
- Execute bit: the CPU can only fetch kernel-mode code when this bit is set to 1. However, the CPU is still allowed to fetch user-mode code even when this execute bit is set to 0 (i.e., user-mode code execution is enforced by page permission set by the OS).
- Write bit: If this bit is set to zero, even the OS cannot write to this memory region, regardless of page permissions.
- Locked bit: After this bit is set, the operating system cannot make any modifications to this permission entry.

- Valid bit (optional): this indicates a permission entry has been fully initialized. Writes to the permission registers may not be atomic; hence this bit can be used to make it easier for the software to prevent the hardware from enforcing a permission entry before it is fully initialized. Note that enforcing a partially initialized permission may prevent the operating system from resuming execution.

The hardware consults permissions in this table on each L1 cache access. If there are overlapping address ranges in the list of permissions, we enforce the most restrictive ones.

Register Locking. As mentioned above, to prevent any additional modifications to a permission entry or a configuration status register, the kernel can set the lock bit associated with that register. We can use the lock bit to mask the "write-enable" input of the register. This mechanism is shown in Figure 7.2. The only way to reset these registers once they are locked is to reboot the system.

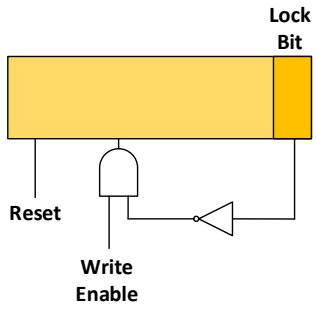


Figure 7.2 Register Locking. The lock bit can be used to mask the write enable input. Once locked, the register can only be overwritten after resetting the system.

7.3.1 Eliminating Additional Memory Latency

The kernel integrity enforcement scheme we are proposing requires a permission check operation on every data and instruction fetch. Therefore, it is essential that the permission lookup process does not stall the CPU lest it will incur overheads on every cache read operation.

Keeping Permission Tables Small To do quick permission lookups, we need to keep the hardware table small. Searching through a hardware maintained table will be slower and more expensive (in terms of energy consumption and silicon cost) as the size of the table grows.

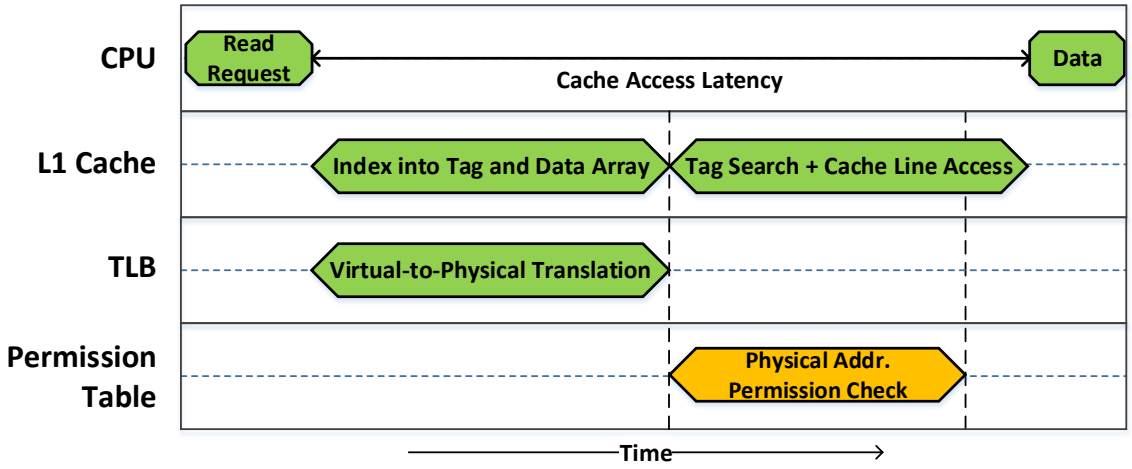


Figure 7.3 Performing Permission Check in Parallel with Cache Access. After the virtual-to-physical address translation is completed, performing tag search and cache line access requires additional cycles (typically ~ 4 cycles). This time window can be used to perform permission checks without impacting the cache access latency. The above figure illustrates the case where we have both an L1 cache hit and a TLB hit.

If the security-critical kernel memory regions are spread across numerous physically non-contiguous memory regions, then we will need a large permission table to protect them. To protect the operating system using a small, high-speed permission table, we make minor modifications to the kernel to store the regions that need to be protected by Neverland within a limited number of physically contiguous memory locations. These software modifications are discussed later in Section 7.4. In our evaluations, an 8-entry table was quite sufficient to protect all critical memory regions in the Linux kernel. Our evaluations also show that such small tables can be integrated into high-frequency CPUs without impacting cycle-time, and can be searched within 3 CPU cycles (Section 7.7.1).

Parallelizing Permission Lookups The permission checks can be timed in a manner that ensures the extra cycles that are required to search the table do not stall the CPU. Figure 7.3 illustrates the operations involved in a cache access. Initially, the CPU issues a virtual address it wants to access. This address is translated to a physical address using the translation lookaside buffer (TLB). On high-performance CPUs, portions of virtual address bits are used to initiate cache access – in parallel with the address translation

process (more formally, the virtual address is used to index into the tag and data arrays). Once the address translation process is done, the physical address is used to search the tags and select the appropriate data array.

As it can be seen in the timing diagram in Figure 7.3, once the physical address is returned by the TLB, there is time window at the end in which the cache performs tag comparisons and reads the appropriate cache line. The physical memory permissions can be read in parallel with this last operation – without stalling the CPU. On today's high-performance CPUs, the tag-compare and data-access operations in the L1 cache take $\sim$ 4 CPU cycles [39]. The results we present in Section 7.7 show that this 4-cycle time window is enough to perform physical address permission lookups in parallel. As a result, Neverland's protections do not incur any performance overheads.

On CPUs without virtually indexed caches (i.e., that do not overlap cache and TLB lookup), it is relatively more straightforward to do permission lookup in parallel with cache access as the tag-search and data-array-select operations take even longer.

7.4 Supporting Loadable Kernel Modules

Modern operating systems need to support a vast array of platforms and peripherals. Compiling the kernel with all the drivers or OS extensions that will ever be required would result in an extremely bloated operating system.

To that end, operating systems typically allow drivers to be implemented as separate kernel modules that can be dynamically linked to the kernel as necessary. The core kernel loads all the modules that are required to run the OS on the target hardware configuration. On a typical Linux machine, for example, there could be 10s or 100s of kernel modules that need to be loaded into the system (on a laptop we edited this chapter on, for example, a total of 171 kernel modules were automatically loaded at boot time).

Loadable kernel modules present a challenge to Neverland's memory protection mechanisms. In addition to protecting the core kernel, Neverland needs to protect the text (code) section and security-critical data of all loaded kernel modules – **which**

are going to be loaded at arbitrary, physically non-contiguous memory addresses. This fragmentation is illustrated in Figure 7.4. Each of the potentially 100s of modules loaded into the system could have code and data that span multiple virtual pages that are *not* physically contiguous.

If Neverland ends up maintaining a large table with 100s of entries, then the permission lookup operation will be slow – which incurs performance overheads. Furthermore, having large permission tables will be impractical in CPUs used by mobile and embedded devices – due to their cost and energy constraints.

Rearranging Physical Memory Layout To enable Neverland to support any number of kernel modules in a scalable manner, we need to modify the kernel module loader logic. In this work, we modified the loadable kernel loader in the Linux kernel so that all text (code) sections from loadable kernel modules will be laid out on a physically contiguous memory region. Although the modifications and results we present in this work are in the context of the Linux kernel, the algorithm presented here can be adopted into any other OS. Furthermore, the same algorithm can be used to contiguously store memory allocated by any kernel subsystem.

The steps taken by our algorithm for laying out kernel modules is illustrated in Figure 7.4 and is summarized below:

- **Step 1:** ***Allocate a large physically contiguous region*** before any kernel module is loaded (or at boot time). The size of this contiguous region can be configured as a kernel parameter, and can be made arbitrarily large since any unused portions will be freed at the end.
- **Step 2:** ***for each loadable kernel module***:
  - **Step 2.1:** ***Load kernel modules using the regular flow.*** We do not need to modify the core logic for linking and loading kernel modules.
  - **Step 2.2:** ***Copy the content of the pages we want to protect*** to the physically contiguous region.

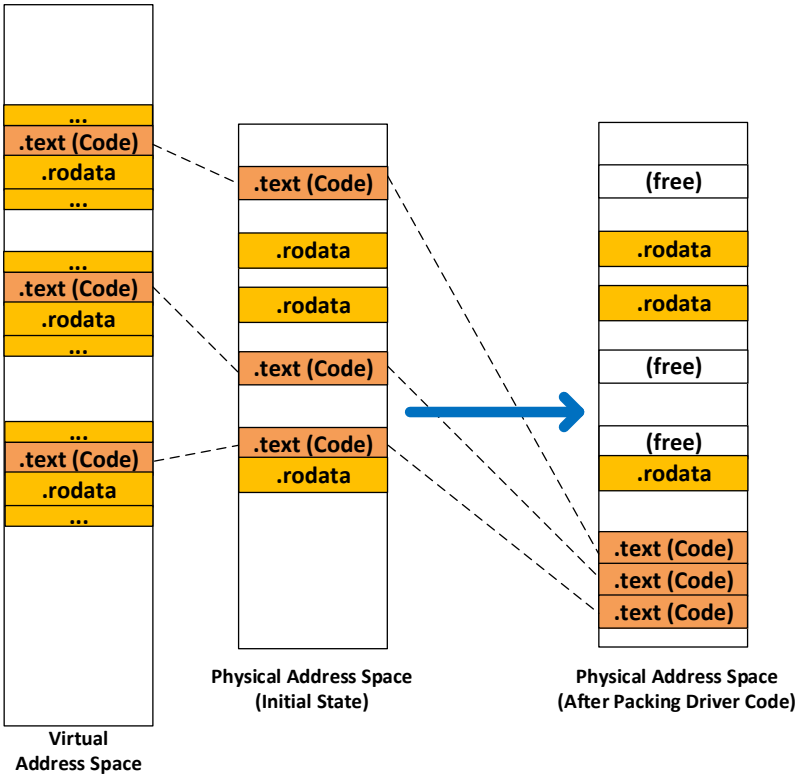


Figure 7.4 "Defragmenting" Kernel Module Code. The code sections of loadable kernel modules are placed in numerous physically non-contiguous memory regions. As a result, protecting them directly would require numerous entries in the permission table. To avoid the need for large hardware tables, we re-arrange the physical memory layout and place all the kernel modules' code sections in a physically contiguous region. This enables us to protect all code sections using a single entry in the permission table. The algorithm we present in Section 7.4 can be used to "defragment" any kernel-mode code/data that needs to be protected by Neverland.

  - **Step 2.3:** ***Overwrite page table entries (PTEs)*** for the copied pages so that the PTEs point to a physical address in the physically contiguous region.
  - **Step 2.4:** ***Flush TLB*** to ensure the hardware will see the modifications to the kernel's PTEs.
  - **Step 2.5:** ***Free the old physical page frames.***

- **Step 3:** ***Free any unused memory from the physically contiguous pool***, after all modules have been loaded.

At the end of the above operations, the *virtual address* space remains unchanged, and as a result, the rest of the kernel does not need to be modified to support these changes.

As we will detail in the next section, all loadable kernel modules and drivers will have to be loaded at boot time for the system to fully benefit from Neverland's protections.

7.5 Restrictions on Kernel Features

To benefit from the maximum security guarantees afforded by Neverland, the kernel must mark all privileged code read-only, and also lock the permission table entries. Such tight restrictions can go against some legitimate system features. In this section, we identify four such features and explain how they are affected by Neverland.

Runtime Kernel Module and Driver Loading. Driver software and kernel modules are typically loaded into a system at boot time. Most operating systems, however, allow drivers or kernel modules to be loaded at any arbitrary time after the system is done booting.

To allow drivers to be loaded anytime while the OS is running, Neverland's "non-privileged-code-only" restrictions on memory regions must be disabled. This weakens the security guarantees provided by Neverland. Therefore, to ensure maximum security, it is desirable to require all necessary kernel modules to be loaded at boot time, and

then lock all privileged code regions from additional modification. With this restriction, the system must be restarted when a driver needs to be updated or installed.

Requiring a restart on OS or driver update will only be a slight inconvenience for regular users, especially on mobile and desktop devices, as all necessary drivers and kernel modules are typically loaded at boot time; and restarting the system on OS and driver updates is already a common practice. On Android systems, for example, it is already a recommended practice to load all required kernel modules in one pass when the kernel is initialized (for improved performance) [3].

Self-Modifying Code and Just-in-Time Compilation. Most modern CPUs allow pages to be marked non-executable. A typical page that contains code is marked read-only and executable, whereas all other pages are set to be non-executable.

However, supporting self-modifying code or just-in-time compilers requires code pages to be writable at least temporarily – which can potentially expose the system to code injection attacks. To protect a system from injection of kernel-mode code, a Windows 10 system protected by Microsoft's Device Guard does not allow drivers to dynamically allocate executable pages [126]. Neverland's protections can be enabled on such restrictive systems without requiring significant modifications.

Modern Linux distributions, however, support an in-kernel just-in-time compiler, known as eBPF (extended BSD Packet Filter), that enables user space programs to run sandboxed bytecode in kernel-mode. This feature is an extension of the original BSD Packet Filter system that enables high-speed packet filtering by avoiding expensive copies and context switches between user and kernel-mode [60].

In-kernel JITs such as eBPF, cannot properly function with the strict configuration presented in the previous sections. However, some of these restrictions can be relaxed to allow in-kernel JITs. Instead of making *all* privileged-code regions unwritable, Neverland can be configured to allow writes to a fixed, pre-allocated executable memory region that is used by the in-kernel JITs *only*. All other privileged code (core kernel and driver code) will still be unwritable until a system reset.

It is conceivable that an attacker could find a bug that would allow code-injection into the fixed memory region that is explicitly pre-allocated for the in-kernel JIT. However,

even with only portions of the privileged code regions marked as write-once, Neverland would still make privilege escalation attacks significantly harder compared to a system that only relies on page permissions.

Live Kernel Patching. To avoid the need to restart a server on an OS update, some operating systems support live kernel patching (also called hot patching) – whereby parts of the kernel code can be overwritten without restarting the system [48, 126]. The protections introduced in this paper, by design, disallow modification of kernel code once the boot process is done. This effectively makes live kernel patching illegal.

On a cloud deployment, the hypervisor and/or host OS can be protected using Neverland, and still allow guest operating systems to be live patched. Patching a Neverland protected kernel that is directly running on the hardware, however, will require a system reboot.

Kernel live patching is *not* a critical feature on desktop and mobile systems, and as such Neverland's protections will not introduce significant usability issues.

In Summary... Neverland's full kernel protections can be readily adopted on mobile, embedded, and desktop platforms as these machines do not need to load new drivers frequently, do not need to rely on in-kernel JITs, and can usually be restarted when the OS or drivers need to be updated.

On certain server deployments, however, enabling the full set of defenses we presented would require either disabling features such as live kernel patching and privileged JITs, or relaxing some of the restrictions (e.g., allowing the OS to update some privileged code regions).

7.6 Security Analysis

As discussed in Section 6.2, numerous defenses have been proposed to harden operating systems against attacks. Among these defenses, Neverland requires secure boot and driver signature verification to ensure that a tampered kernel or a malicious driver

Attacks (see Section 6.1)	**Protection by Neverland**
Syscall Table (SSDT) and Interrupt Descriptor Table (IDT) Hooking [41]	Full Protection: tables unwritable after boot
Configuration (Model Specific) Register Hooking [41]	Full Protection: registers that hold system call and interrupt entry points are locked after boot
Runtime Code Patching [41]	Full Protection: text sections are locked; no privileged code can exist outside of the locked area
Code Pointer Hooking [123]	Limited Protection: kernel pointers that need to change overtime cannot be locked; an attacker can hook these pointers, but they can only point to existing kernel code (i.e., cannot point to new attacker injected code).
Direct Kernel Object Manipulation (DKOM) [41]	Limited Protection: objects that need to be updated over time cannot be locked; an attacker can use code-reuse attacks or exploit memory management bugs to overwrite objects.
Malicious Drivers [41]	Full Protection: no new privileged code can be loaded after boot
Privilege Escalation (User space to Kernel space) (§6.1)	Full Protection: no code outside the locked region can execute with kernel privileges

Table 7.1 Effectiveness of Neverland's Protections. Neverland's hardware controlled memory permissions are effective at a protecting against most OS attack vectors. Code pointer hooking and direct kernel object manipulation (via code-reuse attacks), however, are not thwarted by Neverland.

is not loaded before the memory permission tables are initialized. Neverland cannot provide reliable protection if the attacker compromises the secure boot chain before the permission table is initialized.

Table 7.1 highlights how Neverland helps protect against existing attack vectors and stealth techniques. It can be seen from the table that Neverland's hardware controlled memory permissions are effective at thwarting most attack vectors. The two exceptions

are code pointer hooking and direct kernel object manipulation. Indefinitely locking arbitrary code pointers and kernel objects from modifications is not feasible since the operating system might need to update them as the state of the system changes. As a result, an attacker could still hook code pointers or leverage code-reuse techniques such as return-oriented-programming [103] to maliciously modify kernel objects on a system protected by Neverland.

Even if Neverland cannot be used to directly protect dynamic kernel objects and pointers, the fact that it can totally disable any unauthorized privileged execution makes it hard to meaningfully manipulate these unprotected pointers and data structures. The attacker will need to purely rely on code-reuse attacks to modify desired memory regions. Furthermore, after hooking code pointers, the attacker cannot redirect them to new privileged code, but only to existing kernel/driver code. Operating systems can also use Neverland to "lock" kernel configuration flags, such as driver signature enforcement configurations – which are data structures typically targeted by kernel exploits [19, 31]. Hence, the protections we have presented in this work significantly limit what the attacker can achieve by way of kernel exploits.

7.7 Evaluation

We evaluated the effectiveness and practicality of Neverland by adding the defenses presented above to a RISC-V system. We extended Spike, the official RISC-V ISA emulator [92], with the proposed hardware extensions. We modified the Linux kernel to i) initialize the permission tables, ii) lock the configuration registers which store the addresses of the trap handlers (mtvec and stvec registers on RISC-V), and iii) defragment loadable kernel modules. RISC-V already has an (optional) ISA extension for physical memory protection [115] – which is an approach typically used as a replacement to page-based permissions on low-resource microcontrollers [11, 24]. Hence, we use those instructions to write to the permission registers. On other architectures, the registers in the permission table could be programmed through a memory-mapped I/O.

We were able to boot a fully functional 64-bit Linux-based system with BusyBox 1.26.2 utilities [1] on the Spike emulator with these protections in place. This validates

that a stock kernel (with minor modifications) can run with portions of its memory completely locked-down by the hardware. We initialize the permission table once the kernel is fully initialized and read-only page permissions are set up by the kernel. In the remainder of this section, we evaluate the overheads associated with these changes.

7.7.1 Hardware Overhead

The only additional hardware that is necessary for locking-down memory locations is the permission table and the permission check hardware.

To measure permission lookup latency and the hardware cost, we implemented a permission table and lookup logic in RTL and used the Synopsis Design Compiler to synthesize the designs to the IBM 45nm silicon-on-insulator (SOI) technology library.

Speed. We were able to clock our synthesized permission table at frequencies as high as 3.83 GHz on a 45nm library. For comparison, the highest peak frequency available on a 45nm Intel CPU is 3.73GHz [25]. This confirms permission tables can be integrated even in high-frequency CPUs without affecting speed. 45nm is a relatively older technology node, but this comparison holds true on newer silicon technology as well since both the permission table and rest of the CPU scale in a similar manner.

Furthermore, using this design, it is possible to perform a lookup in a 16-entry permission table in just 3 cycles. And as described in Section 7.3.1, this 3 cycle table lookup can be performed in parallel with the tag comparison and data array access – without incurring any performance overhead.

Hardware Area and Energy. Table 7.2 lists the estimated power and area overheads of permission tables synthesized with different sizes. We present the overheads normalized with respect to i) a 40nm, dual-core ARM Cortex-A9 (a CPU targeted at low-power, cost-sensitive embedded systems [72]), ii) a single-core 45nm, RISC-V BOOM with no L2 cache (a CPU that targets similar device classes as the Cortex-A9), and iii) a 45nm, Intel Core i5 (a desktop-class CPU). The power and area of the two baseline embedded/mobile CPUs is based on the data from [22], while the corresponding

No. of Entries (per core)	Normalized Area Overhead			Normalized Power Overhead		
	BOOM	Cortex-A9	Core i5	BOOM	Cortex-A9	Core i5
4 x 2	0.67%	0.59%	0.016%	4.29%	1.79%	0.07%
8 x 2	1.26%	1.11%	0.032%	7.69%	3.20%	0.13%
16 x 2	2.39%	2.10%	0.062%	14.18%	5.91%	0.23%

Table 7.2 Power and Area Overhead of Permission Tables. Overheads normalized with respect to a 40nm dual-core ARM Cortex-A9 (a CPU targeted at low-power, cost-sensitive embedded systems), a single-core 45nm RISC-V BOOM CPU with no L2 cache, and a 45nm quad-core desktop-class Intel Core i5-760 CPU. The estimates for the mobile CPUs are pessimistic as the permission tables are not synthesized to a power-optimized process node.

estimates for the permission tables are based on a 45nm synthesis result at a target frequency of 1.5GHz. The baseline power and die size for the desktop-class CPU was acquired from the Core i5-760 datasheet, while its corresponding permission table is synthesized with a target frequency of 3.8GHz. The estimates account for 2 permission tables *per core* – one for the I-cache and a second one for the D-cache.

The results show that adding 8-entry permission tables even on low-end embedded CPUs such as the Cortex-A9 incurs minimal overhead – 1.11% area overhead and 3.2% power overhead. These estimates are pessimistic as the permission tables are not synthesized to energy optimized process. These overheads will be even lower on CPUs that do not merely target low-end devices. On smaller CPUs such as the BOOM or Cortex-A5, adding more than 4-entry permission tables (without power optimizations) would result in a sizable power overhead.

7.7.2 Performance Overhead

The evaluation above indicates that permission lookups, which are performed on each L1 cache access will not degrade system performance. In addition to the hardware extension, however, we made two additions to the kernel code, namely packing the text section of loadable kernel modules and setting up the permission table at boot time. We evaluate the performance impact of these kernel modifications on boot latency below. Note that these two modifications only impact the boot latency as they only need to be executed once on system startup.

Code Size	Module Load Time		
	Baseline	Packed Text Section	Overhead
128KB	4.86 ms	5.89 ms	1.03 ms
256KB	5.15 ms	6.27 ms	1.13 ms
512KB	5.91 ms	7.28 ms	1.37 ms
1MB	7.32 ms	8.67 ms	1.35 ms

Table 7.3 Module Load Latency Overhead Incurred by Defragmentation.

Kernel Module Loading

At the time of this project, the upstream RISC-V Linux port (v. 4.14) did not yet fully support dynamically loadable kernel modules. Hence, we validated our kernel module packing scheme and characterized its performance on an x86_64 system. We implemented the algorithm described in Section 7.4 in version 4.14 of the kernel, and run it on a KVM virtual machine.

To measure the additional latency incurred on kernel module loading as a result of moving code pages, we compiled and loaded kernel modules with different code sizes. The results are presented in Table 7.3. The results demonstrate that relocating even 100s of KBs of code pages incurs $< 1.5ms$ overhead.

Initializing Permission Registers

After the boot process is completed (including kernel module packing), we need to write the appropriate permissions to the hardware table. When booting the Linux kernel on a single emulated core in the Spike emulator, writing the addresses and permissions to the hardware table only introduced an additional $270\mu s$ latency on the boot process. We expect the latency to be even lower on real hardware that does not have the emulation overhead. Note that the initialization needs to be done on every core on a multicore system – similar to any other configuration status registers.

Chapter 8

Conclusion

This dissertation presented a set of mechanisms and optimizations for ensuring memory system security at low overheads. In the first part of this dissertation, we illustrated that physical attacks continue to be practical even on today's scrambled memory systems, and proposed approaches for lowering the overheads associated with strong memory encryption and authentication.

In the second part of this dissertation, we presented a hardware mechanism for thwarting kernel-mode malware and rootkits. Our approach prevents runtime modification of the kernel's critical code/data by locking them at the hardware level once the boot process is complete.

The mechanisms and optimizations proposed in this dissertation can serve as *low overhead* building blocks towards creating a more secure computing environment that can be used to deploy security-critical applications.

8.1 Future Research Directions

A number of additional challenges remain in preventing physical attacks on off-chip memory and in safeguarding integrity of kernel memory. These problems require additional research beyond what is presented in this dissertation. We highlight some of these challenges in this section.

Challenges in Efficient Secure Off-Chip Storage

The optimizations presented in Part 1 of this dissertation significantly lower the storage overheads associated with authenticated memory encryption, and have the potential to lower the barrier to deploying such strong protections. However, supporting authenticated memory encryption on resource-constrained embedded devices that need to access off-chip memory remains challenging due to the additional hardware that is necessary. Furthermore, integrating authenticated memory encryption with a trusted execution environment that provides full-fledged software attestation and isolation (similar to Intel SGX), requires even more on-chip resources. Additional explorations are required to reduce this cost and to make these defenses suitable for resource-constrained microcontrollers.

Challenges in Securing Privileged Software

Protecting Dynamic Kernel Data. Taking away the ability to execute malicious kernel-mode code or to ability modify static kernel data significantly limits the attacker's facilities. Creating a full-fledged kernel-mode malware or rootkit by relying entirely on code-reuse attacks and kernel object manipulation is challenging. Even so, attackers can still leak sensitive data, or escalate a user's or processes' permissions via code-reuse attacks and direct-kernel object manipulation. Since the protections presented in Chapter 7 are not suitable for protecting dynamic kernel data, an additional layer of defense is necessary for protecting critical kernel objects.

Securing System Management Mode Code. In addition to kernelspace and userspace programs, today's CPUs run firmware that is responsible for tasks such as power management and hardware initialization. These firmware typically run with higher privileges than the OS – at privilege level referred to as System Management Mode (SMM) or ring -2 on Intel systems. Code running in SMM can access the entire system RAM. On the other hand, the operating system is restricted by the hardware from accessing the SMM firmware's memory. As a result, an attack that can penetrate into these firmware would have unrestricted access to the full system and would be extremely

hard to detect. Such attacks have been practically demonstrated [27, 49, 28, 119], underscoring the need to safeguard these highly privileged firmware. Additional exploration is required to extend the OS kernel protections we present in this dissertation to reliably defend the integrity of the system management firmware.

Server Deployments. Making Neverland usable in server environments requires additional studies. For example, as described in Chapter 7, restrictions imposed by Neverland's protections make it impossible to perform live kernel patching – a feature that is valuable for servers that need to apply OS patches without being restarted.

Finally, extending Neverland to improve the security of hypervisors running on servers that host multiple virtualized guest operating system is another promising area for further exploration. To be practical, the hardware extensions safeguarding the hypervisor's memory must be able to retain the system's ability to start and stop virtual machines on-demand.

Appendix

Memory Scrambler Initialization

One of the features that makes cold attacks challenging on late-model Intel-based DDR3 and DDR4 machines is that the key used to scramble memory contents changes after each reboot. Keys extracted for one boot session cannot be used to descramble memory contents on future boot sessions. We start our analysis by inspecting how this randomization is performed.

By studying the BIOS firmware for an Intel SandyBridge machine [2], we were able to discover that the memory scrambler is initialized with a new 32-bit seed during the boot process. However, the same seed is reused if a system is woken up from S3 sleep mode - this is necessary to ensure that data written before sleep mode can be read afterwards. For this reason, the seed that is generated during boot is saved to a non-volatile memory until the system wakes. In one of the SandyBridge machine we analyzed, we were able to access the stored scrambler S3 seed and overwrite it with any value using the *nvramtool* utility.

In addition to the S3 seed, the firmware also saves a second seed. To understand to importance of the second seed, we analyzed the scrambler output after overwriting flash memory with different random seeds. Our results suggest that the second seed is a pre-generated random number to be used for the next boot session. In other words, the seed that the system will use for the next boot session can be known by the attacker, if one has access to non-volatile memory. Furthermore, a new seed is generated during the next boot session by using the current seed as an input to the random number generator. As a result, we observed the same succession of seeds being generated after cold reboot if we reset the contents of the BIOS memory to contain a specific seed.

Under this scheme, note that the seed that is used during this boot session will be registered as the S3 seed and then overwritten by a newly generated seed.

We observed that the systems we analyzed reset the seed to an new random value, instead of using the pre-generated seed if:

1. A DRAM module is replaced while the system is turned off (memory controllers can probe details of the DDR modules using a standardized mechanism).

2. The checksum computed over the S3 and the cold boot seeds and stored in the BIOS memory is incorrect.

3. The seed in the non-volatile memory is overwritten to zero.

The flow chart in Figure A.1 summarizes the characteristics of the random number generator. The observation that a seed leaves the CPU chip and is stored in an unsecured non-volatile memory poses security risks. An attacker can potentially exploit this hole by using the right equipment to extract seeds from the flash memory. However, in our proof-of-concept attack, we will demonstrate that the scrambler can be bypassed without any knowledge of the seed.

Key Idea: A new seed is used to initialize the memory scrambler during each cold boot. In one of our test machines, we were able use software tools to read and manipulate the seeds in the BIOS without much difficulty.

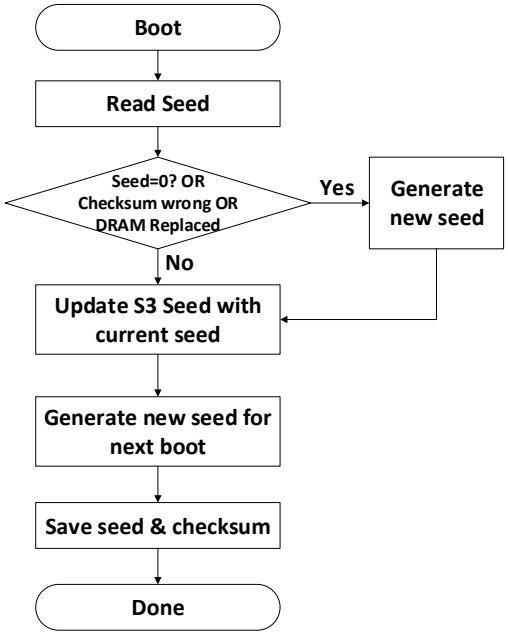


Figure A.1 Scrambler Seed Generator Logic.

Bibliography

[1] BusyBox: The Swiss Army Knife of Embedded Linux. https://busybox.net/. Accessed: August 2018.

[2] coreboot Open Source Firmware. https://www.coreboot.org/. Accessed: February 2016.

[3] Modular Kernel Requirements. https://source.android.com/devices/architecture/kernel/modular-kernels. Accessed: August 2018.

[4] JEDEC Solid State Technology Association and others. JEDEC Standard: DDR4 SDRAM. *JESD79-4*, 2012.

[5] JEDEC Solid State Technology Association and others. DDR4 SDRAM SO-DIMM Design Specification. *JEDEC Standard No. 21C*, 2016.

[6] A. Allievi. Understanding and Defeating Windows 8.1 Kernel Patch Protection. *NoSuchCon*, 2014.

[7] I. Anati, S. Gueron, S. Johnson, and V. 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 (WASP)*, 2013.

[8] R. J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Publishing, 2 edition, 2008.

[9] Apple Inc. System Integrity Protection Guide. https://developer.apple.com. Accessed: August 2018.

[10] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, et al. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[11] Z. B. Aweke and T. M. Austin. uSFI: Ultra-Lightweight Software Fault Isolation For IoT-class Devices. *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018.

[12] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision Across Worlds: Real-Time Kernel Protection From The Arm Trustzone Secure World. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014.

[13] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. Hypersentry: Enabling Stealthy In-Context Measurement Of Hypervisor Integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2010.

[14] J. Bauer, M. Gruhn, and F. C. Freiling. Lest We Forget: Cold-Boot Attacks on Scrambled DDR3 Memory. *Digital Investigation*, 2016.

[15] G. Beniamini. QSEE Privilege Escalation Vulnerability And Exploit (CVE-2015-6639). `http://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html`. Accessed: August 2018.

[16] G. Beniamini. Trust Issues: Exploiting TrustZone TEEs. `https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html`, 2017. Accessed: August 2018.

[17] D. J. Bernstein. ChaCha, A Variant of Salsa20. In *Workshop Record of SASC*, 2008.

[18] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2008.

[19] A. Blaich, M. Bazaliy, and S. Hardy. Mobile Espionage in the Wild: Pegasus and Nation-State Level Attacks. In *BlackHat Europe*, 2016.

[20] Y. Bulygin, J. Loucaides, A. Furtak, O. Bazhaniuk, and A. Matrosov. Summary of Attacks Against BIOS and Secure Boot. *DefCon*, 2014.

[21] E. Bursztein. Speeding up and Strengthening HTTPS Connections for Chrome on Android. `https://googleonlinesecurity.blogspot.com/2014/04/speeding-up-and-strengthening-https.html`. Accessed: August 2016.

[22] C. Celio, K. Asanovic, and D. Paterson. The Berkeley Out-of-Order Machine(BOOM!): An Open-Source Industry-Competitive, Synthasizable, Parameterized RISC-V Processor. *Third RISC-V Workshop*, 2016.

[23] P. M. Chen and B. D. Noble. When Virtual Is Better Than Real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*. IEEE, 2001.

[24] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer. Protecting Bare-Metal Embedded Systems with Privilege Overlays. *IEEE Symposium on Security and Privacy (SP)*, 2017.

[25] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz. CPU DB: Recording Microprocessor History. *ACM Queue*, 10(4), 2012.

[26] J. J. Drake, Z. Lanier, C. Mulliner, P. O. Fora, S. A. Ridley, and G. Wicherski. *Android Hacker's Handbook*. John Wiley & Sons, 2014.

[27] L. Duflot, D. Etiemble, and O. Grumelard. Using CPU System Management Mode To Circumvent Operating System Security Functions. *CanSecWest*, 2006.

[28] S. Embleton, S. Sparks, and C. C. Zou. SMM Rootkit: A New Breed Of OS Independent Malware. *Security and Communication Networks*, 2008.

[29] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[30] S. Field. Microsoft Azure Uses Error-Correcting Code Memory for Enhanced Reliability and Security. `https://azure.microsoft.com/en-us/blog/microsoft-azure-uses-error-correcting-code-memory-for-enhanced-reliability-and-security/`. Accessed: August 2017.

[31] G DATA Security. Uroburos – Deeper Travel Into Kernel Protection Mitigation. `https://www.gdatasoftware.com/blog/2014/03/23966-uroburos-deeper-travel-into-kernel-protection-mitigation`. Accessed: July 2018.

[32] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *NDSS*, 2003.

[33] B. Gassend, G. E. Suh, D. Clarke, M. Van Dijk, and S. Devadas. Caches and Hash Trees for Efficient Memory Integrity Verification. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2003.

[34] X. Ge, H. Vijayakumar, and T. Jaeger. Sprobes: Enforcing Kernel Code Integrity on the TrustZone Architecture. In *Proceedings of Mobile Security Technologies (MoST) workshop*, 2014.

[35] M. Gruhn and T. Muller. On the Practicability of Cold Boot Attacks. In *Proceedings of the Eighth IEEE International Conference on Availability, Reliability and Security (ARES)*, 2013.

[36] S. Gueron. Intel's SGX Memory Encryption Engine. *Proceedings of the Real World Cryptography Conference*, 2016.

[37] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold-Boot Attacks on Encryption Keys. *Communications of the ACM*, 52(5), 2009.

[38] N. Hassan and R. Hijazi. *Digital Privacy and Security Using Windows: A Practical Guide*. Apress, 2017.

[39] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.

[40] M. Henson and S. Taylor. Memory Encryption: A Survey of Existing Techniques. *ACM Computing Surveys (CSUR)*, 2013.

[41] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2006.

[42] A. Honig and N. Porter. 7 Ways We Harden Our KVM Hypervisor at Google Cloud: Security in Plaintext. `https://cloudplatform.googleblog.com/2017/01/7-ways-we-harden-our-KVM-hypervisor-at-Google-Cloud-security-in-plaintext.html`. Accessed: August 2017.

[43] H. Hsing. AES Core. `http://opencores.org/project,tiny_aes`, 2013. Accessed: August 2016.

[44] A. Huang. *Hacking the Xbox:An Introduction into Reverse Engineering.* No Starch Press, 2003.

[45] R. C. Huang and G. E. Suh. IVEC: Off-Chip Memory Integrity Protection for Both Security and Reliability. In *International Symposium on Computer Architecture (ISCA)*, 2010.

[46] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion. SoK: Introspections on Trust and the Semantic Gap. *IEEE Symposium on Security and Privacy*, 2014.

[47] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007.

[48] S. J. Josh Poimboeuf. Introducing kpatch: Dynamic kernel patching. `https://rhelblog.redhat.com/2014/02/26/kpatch/`, 2014. Accessed: August 2018.

[49] C. Kallenberg, X. Kovah, J. Butterworth, and S. Cornwell. Extreme Privilege Escalation On Windows 8/UEFI Systems. *BlackHat*, 2014.

[50] D. Kaplan, J. Powell, and T. Woller. AMD Memory Encryption Whitepaper, 2016.

[51] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. ret2dir: Rethinking Kernel Isolation. In *USENIX Security Symposium*, 2014.

[52] L. Koromilas, G. Vasiliadis, E. Athanasopoulos, and S. Ioannidis. GRIM: Leveraging GPUs for Kernel Integrity Monitoring. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016.

[53] D. Kwon, K. Oh, J. Park, S. Yang, Y. Cho, B. B. Kang, and Y. Paek. Hypernel: A Hardware-Assisted Framework for Kernel Protection Without Nested Paging. In *Proceedings of the 55th Annual Design Automation Conference (DAC)*. ACM, 2018.

[54] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. B. Kang. KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object. In *USENIX Security Symposium*, 2013.

[55] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *ASPLOS*, 2000.

[56] S. Lindenlauf, H. Hofken, and M. Schuba. Cold Boot Attacks on DDR2 and DDR3 SDRAM. In *Proceedings of the 10th IEEE International Conference on Availability, Reliability and Security (ARES)*, 2015.

[57] Z. Liu, J.-H. Lee, J. Zeng, Y. Wen, Z. Lin, and W. Shi. CPU Transparent Protection of OS Kernel and Hypervisor Integrity with Programmable DRAM. In *International Symposium on Computer Architecture (ISCA)*, 2013.

[58] A. V. Luța. From Ring3 To Ring0: Exploiting The Xen x86 Instruction Emulator. `labs.bitdefender.com`, 2014. Accessed: July 2018.

[59] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: Practical Oblivious Computation in a Secure Processor. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2013.

[60] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *USENIX Winter Conference*, 1993.

[61] J. Meza, Q. Wu, S. Kumar, and O. Mutlu. Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field. In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2015.

[62] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang. Vigilare: Toward Snoop-Based Kernel Integrity Monitor. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2012.

[63] P. Mosalikanti, C. Mozak, and N. Kurd. High Performance DDR Architecture in Intel Core™ Processors Using 32nm CMOS High-K Metal-Gate Process. In *Proceedings of the IEEE International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 2011.

[64] T. Müller, F. C. Freiling, and A. Dewald. TRESOR Runs Encryption Securely Outside RAM. In *Proceedings of the 20th USENIX Conference on Security*, 2011.

[65] T. Müller and M. Spreitzenbarth. FROST: Forensic Recovery of Scrambled Telephones. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, 2013.

[66] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSS: A Full System Simulator for Multicore x86 CPUs. In *Proceedings of the 48th Design Automation Conference*. ACM, 2011.

[67] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *IEEE Symposium on Security and Privacy*. IEEE, 2008.

[68] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012.

[69] P. Pessl, D. Gruss, C. Maurice, and S. Mangard. Reverse Engineering Intel DRAM Addressing and Exploitation. *CoRR*, abs/1511.08756, 2015.

[70] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - A Coprocessor-Based Kernel Runtime Integrity Monitor. In *USENIX Security Symposium*, 2004.

[71] Amazon Web Services. Amazon EC2 FAQs. `https://aws.amazon.com/ec2/faqs/`. Accessed: August 2017.

[72] ARM Holdings. ARM Cortex-A9 . `https://developer.arm.com/products/processors/cortex-a/cortex-a9`. Accessed: July 2018.

[73] ARM Holdings. Building a Secure System using TrustZone Technology, 2005.

[74] ARM Holdings. ARMv7-M Architecture Reference Manual, 2014.

[75] ARM Holdings. *ARM Cortex-A Series Programmer's Guide for ARMv8-A*, 2015.

[76] Hybrid Memory Cube Consortium. *Hybrid Memory Cube Specification 1.0*, 2013.

[77] Intel Corporation. *Intel® Atom™ Processor S1200 Product Family for Microserver*, 2012.

[78] Intel Corporation. *5th Generation Intel® Core™ Processor Family, Intel® Core™ M Processor Family, Mobile Intel® Pentium® Processor Family, and Mobile Intel® Celeron® Processor Family Datasheet*, 2015.

[79] Intel Corporation. *6th Generation Intel® Processor Datasheet for S-Platforms*, 2015.

[80] Intel Corporation. *Desktop 4th Generation Intel® Core™ Processor Family, Desktop Intel® Pentium® Processor Family, and Desktop Intel Celeron® Processor Family*, 2015.

[81] Intel Corporation. *ISCA 2015 Tutorial: Intel Software Guard Extensions (Intel SGX)*, 2015.

[82] Intel Corporation. *Intel® 64 and IA-32 Architectures, Software Developer's Manual Volume 3A: System Programming Guide, Part 1*, 2016.

[83] National Vulnerability Database. Vulnerability Summary for CVE-2012-0217. `https://nvd.nist.gov`, 2012.

[84] National Vulnerability Database. Vulnerability Summary for CVE-2013-2596. `https://nvd.nist.gov`, 2013.

[85] National Vulnerability Database. Vulnerability Summary for CVE-2015-6639. `https://nvd.nist.gov`, 2015.

[86] National Vulnerability Database. Vulnerability Summary for CVE-2016-0088. `https://nvd.nist.gov`, 2016.

[87] National Vulnerability Database. Vulnerability Summary for CVE-2016-5195. `https://nvd.nist.gov`, 2016.

[88] National Vulnerability Database. Vulnerability Summary for CVE-2018-0959. `https://nvd.nist.gov`, 2018.

[89] National Vulnerability Database. Vulnerability Summary for CVE-2018-0961. `https://nvd.nist.gov`, 2018.

[90] National Vulnerability Database. Vulnerability Summary for CVE-2018-8219. `https://nvd.nist.gov`, 2018.

[91] National Vulnerability Database. Vulnerability Summary for CVE-2018-8781. `https://nvd.nist.gov`, 2018.

[92] RISC-V Foundation. RISC-V Software Tools. `https://riscv.org/software-tools/`. Accessed: August 2018.

[93] L. Ren, X. Yu, C. W. Fletcher, M. van Dijk, and S. Devadas. Design Space Exploration and Optimization of Path Oblivious RAM in Secure Processors. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[94] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2008.

[95] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin. Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS and Performance Friendly. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (Micro)*, 2007.

[96] D. Rosenberg. QSEE Trustzone Kernel Integer Over Flow Vulnerability. In *Black Hat conference*, 2014.

[97] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters*, 10(1), 2011.

[98] M. E. Russinovich, D. A. Solomon, and A. Ionescu. *Windows Internals, Part 2, 6th ed.* Pearson Education, 2012.

[99] M. Seaborn. How Physical Addresses Map to Rows and Banks in DRAM. `http://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html`. Accessed: February 2016.

[100] M. Seaborn and T. Dullien. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. *Black Hat*, 2015.

[101] F. J. Serna. MS08-061 : The Case Of The Kernel Mode Double-Fetch. `https://blogs.technet.microsoft.com/srd/2008/10/14/ms08-061-the-case-of-the-kernel-mode-double-fetch/`. Accessed: August 2018.

[102] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *SOSP*, 2007.

[103] H. Shacham. The Geometry Of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security*, 2007.

[104] A. Shishkin and I. Smit. Bypassing Intel SMEP On Windows 8 x64 Using Return-oriented Programming. `http://blog.ptsecurity.com/2012/09/bypassing-intel-smep-on-windows-8-x64.html`, 2012. Accessed: July 2018.

[105] P. 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 (ACSAC)*, 2011.

[106] I. Skochinsky. Intel ME Secrets. *Code Blue*, 2014.

[107] S. Skorobogatov. Low Temperature Data Remanence in Static RAM. *University of Cambridge Computer Laborary Technical Report*, 536, 2002.

[108] A. Sodani. Intel® Xeon Phi™ Processor "Knights Landing" Architectural Overview.

[109] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2013.

[110] G. E. Suh, D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (Micro)*, 2003.

[111] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of the 17th Annual International Conference on Supercomputing*, 2003.

[112] J. Szefer, E. Keller, R. B. Lee, and J. Rexford. Eliminating the Hypervisor Attack Surface for a More Secure Cloud. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*. ACM, 2011.

[113] Z. Wang and X. Jiang. Hypersafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *IEEE Symposium on Security and Privacy*, 2010.

[114] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering Kernel Rootkits With Lightweight Hook Protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*. ACM, 2009.

[115] A. Waterman and K. Asanovic. *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture Version 1.10*, 2017.

[116] R. N. Watson. Exploiting Concurrency Vulnerabilities in System Call Wrappers. *Workshop on Offensive Technologies (WOOT)*, 2007.

[117] O. Weisse, V. Bertacco, and T. Austin. Regaining Lost Cycles With HotCalls: A Fast Interface for SGX Secure Enclaves. In *44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017.

[118] R. Wojtczuk. Analysis Of The Attack Surface Of Windows 10 Virtualization-Based Security. *BlackHat*, 2016.

[119] R. Wojtczuk and J. Rutkowska. Attacking SMM Memory via Intel CPU Cache Poisoning. *Invisible Things Lab*, 2009.

[120] W. Xu and Y. Fu. Own Your Android! Yet Another Universal Root. In *Workshop on Offensive Technologies (WOOT)*, 2015.

[121] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin. Improving Cost, Performance, and Security of Memory Encryption and Authentication. *33rd International Symposium on Computer Architecture (ISCA)*, 2006.

[122] J. Yang, L. Gao, and Y. Zhang. Improving Memory Encryption Performance in Secure Processors. *IEEE Transactions on Computers*, 2005.

[123] H. Yin, P. Poosankam, S. Hanna, and D. Song. HookScout: Proactive Binary-centric Hook Detection. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2010.

[124] S. F. Yitbarek, M. T. Aga, R. Das, and T. Austin. Cold Boot Attacks are Still Hot: Security Analysis of Memory Scramblers in Modern Processors. In *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017.

[125] S. F. Yitbarek and T. M. Austin. Reducing the Overhead of Authenticated Memory Encryption Using Delta Encoding and ECC Memory. In *Design Automation Conference (DAC)*, 2018.

[126] P. Yosifovich, D. A. Solomon, and A. Ionescu. *Windows Internals, Part 1: System Architecture, Processes, Threads, Memory Management, and More, 7th ed.* Microsoft Press, 2017.

[127] V. Young, P. J. Nair, and M. K. Qureshi. DEUCE: Write-Efficient Encryption for Non-Volatile Memories. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.