Security Engineering of Hardware-Software Interfaces

Beng Chiew (Adrian) Tang

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2018

ABSTRACT

Security Engineering of Hardware-Software Interfaces

Beng Chiew (Adrian) Tang

Hardware and software do not operate in isolation. Neither should they be regarded as such when securing systems. To seamlessly facilitate computing, they have to communicate via interfaces. Besides characterizing the means by which software can harness the exposed functionalities of hardware, these hardware-software interfaces define the degree and granularity of control and access that software possesses to the lower layers of the system stack. These mechanisms provide a rich source of hardware assistive technologies that can be tapped to enhance security as a full-system property. On the flip side, given the level of access software has to these hardware features, security-oblivious designs of hardware and their interfaces can expose systems to new vulnerabilities. Evidently, these hardware-software interfaces represent a crucial focal area in systems for the formulation, review and refinement of security measures.

This dissertation advances the thesis that security as a full-system property can be improved by examining and leveraging the interworking of hardware and software. It advocates a full-system approach in architecture design by demonstrating how unanticipated ways in which hardware and software co-operate can induce unintended computing behavior and pose security risks. It develops novel techniques to repurpose commodity hardware support to create new defense primitives that exploit the synergy between hardware and software. It shows how commodity hardware-software interfaces play an instrumental role in security with the hardware's well-positioned access to runtime information. All these interface-oriented design principles, as this dissertation demonstrates, are widely applicable and practical as the highlighted three case studies span the three primary stages of a typical security attack, namely the act of inducing unintended system behavior, exploiting vulnerability to achieve initial system control, and executing malicious code for nefarious goals.

First, the dissertation begins by scrutinizing the design of energy management mechanisms, a prevalent class of hardware-software interfaces found in almost all commodity systems. It shows,

for the first time, that as we pursue increasingly aggressive cooperative hardware-software mechanisms to improve energy efficiency, doing so with no regard for security can create serious vulnerabilities. This dissertation highlights a multitude of issues in the current designs of energy management mechanisms. It further demonstrates how, with fine-grained software-based control of underlying hardware voltage and frequency regulators, attackers can exploit these issues to induce unintended computing behavior. It shows that beyond causing unintended system behavior, abusing these interfaces in security-oblivious energy management designs can violate all three key security properties in spite of hardware-enforced isolation: confidentiality (extracting AES keys), integrity (loading self-signed code), and clearly, availability (freezing the device).

Second, the dissertation addresses an advanced class of dynamic code reuse exploits that rely on memory disclosure vulnerabilities to construct their initial payload code at runtime. This class of exploits bypasses even the finest-grained randomization-based defenses. While the concept of execute-only memory in existing defenses works well, it cannot be applied effectively in closed-source systems where perfect disassembly of compiled binaries is not possible. To tackle this problem, this dissertation first introduces the Destructive Code Read primitive—a defense technique that randomizes executable memory as it is being read as data—as a means to thwart memory disclosure exploits as well as to sidestep the problem of imperfect binary disassembly in COTS systems. It leverages the virtualization assistive hardware feature to timely mediate read operations into executable memory, thereby significantly lowering the cost of deploying the Destructive Code Read defense primitive. Tapping into the unique strengths of functionality closer to the hardware layer of the system stack, it extends the benefits of execute-only memory defenses to COTS systems.

Finally, the dissertation builds on the insight that hardware, being the lowest part of the system stack, is uniquely positioned to augment traditionally software-only techniques. Besides being more performant and energy-efficient, hardware offers extensive visibility into code execution at the software layers. This dissertation shows that these hardware characteristics offer unprecedented insights into code execution, both benign and malicious. It demonstrates that the

interaction of hardware and software can be modeled as microarchitectural events, which can in turn be leveraged to detect anomalous malicious code execution in the latter stages of a security attack. Using assistive debugging hardware features to efficiently audit these events, it further develops novel techniques to make sense of the noisy and lower-level microarchitectural events to detect in-flight shellcode execution and full-fledged anomalous malicious programs.

Contents

List of Figures

List of Tables

Acknowledgments

Pursuing a Ph.D. is a significant stage of my life, and is only made worthwhile and rewarding by the people who surround me along the way. For that, I am beyond grateful.

First of all, I have had the immense fortune of being advised by Professor Salvatore Stolfo and Professor Simha Sethumadhavan. Sal is a solid pillar of research inspiration and a fierce advocate of research that makes its way to *practice*. Simha is a bottomless fountain of research optimism and a tireless champion of research that revolutionizes the world. Both of them taught me the value of aiming high and dreaming big. They have been excellent mentors and friends. No page of this dissertation could be possible without them. Thank you Sal and Simha.

I benefited enormously from the knowledge and counsel of many more teachers. First and foremost, I always enjoyed learning from Professor Steven Bellovin as he offered his security insights, peppered with countless interesting stories, sometimes esoteric to say the least. I am also extremely fortunate to work with Professor Suman Jana, as his dedication to concrete details tremendously enriched our research and sharpened my acumen as a researcher. During my stint as the Instructional Assistant (IA) coordinator, Professor Tal Malkin has been a valuable mentor as we navigated the intricate maze of administrative IA matters. I thank Tal for her patient guidance and advice. I would like to also thank Professor Fabian Monrose for all his priceless suggestions and comments that shaped my dissertation to what it is today.

I am indebted to all of my collaborators over the years for all the hard work, brilliant ideas and fun times. John Demme, Matthew Maycock, Jared Schmitz and Adam Waksman took our first

plunge together into the world of hardware-based malware defenses, along with countless rounds of beer at the *1020*. Theofilos Petsio and I tackled the ambitious task of creating the multi-headed *Nezha* bug-finder – Theo is a wonderful friend and the best research (and music) collaborator anyone could ever hope for. Kanad Sinha is my fellow co-cartographer as we painstakingly mapped out and systematized the large body of hardware-support research for software security and expanded our research horizons – I could never hope for a better person to bounce ideas off, or for that matter to debug our *Chi Sao* moves. I have also learned a lot from Professor Mingoo Seok, Professor Jaehyuk Huh, Insu Jang and Sheng Zhang in our collaborations, and I am very grateful.

I owe a debt of gratitude to Professor Michael Sikorski for taking a chance on me as a teaching assistant for his inaugural Malware Analysis and Reverse Engineering class. Michael gave me the much-appreciated opportunity to meld my interest in binary analysis with my love for teaching, and a blast it was for four years! I am also fortunate to work with several very talented people along the way: Nick Harbour, Jonathan Ballone and Matthew Haigh, and I thank them for it.

Thank you to the Columbia undergraduates and exchange students who withstood my mentoring: Abhishek Shah, Kevin Kwan, Alexander Bienstock, Aubrey Douglass Alston, Yifan Lu, Ruoxin Jiang, Joshua Michael Zweig, Kevin Chen, Jennifer Lam. Their potential is limitless.

There are so many more colleagues and ex-colleagues whose presence have been a pillar of support over these years. In no particular order, they are: Fang-Hsiang Su, Hiroshi Sasaki, Yuan Kang, Preetam Dutta, Jill Jermyn, David Williams-King, Sebastian Zimmeck, Naser AlDuaij, Vaggelis Atlidakis, Marios Pomonis, Suphannee Sivakorn, Georgios Argyros, Vasileios P. Kemerlis, Michalis Polychronakis, Miguel Arroyo, Yipeng Huang, Andrea Lottarini, Yinxiao Li, Melanie Kambadur, Lianne Lairmore, John Koh, Chun-Yu Tsai, Ang Cui, Nathaniel Boggs, Peter Du, David Tagatac, Hang Zhao.

Many friends have brightened up my life here in New York and offered the much-needed good fun and distraction from work. I particularly like to thank Jun-Ping Ng and Yan Chen, Jessie Lee and Jason Chen, Thidar Swe Tin and Jochen Weeber, Tsung-Yao Hsu, Sandy Shen, Alan Wu and Vanessa Shyu, Eugene Ang and Rachma Lim.

Much of who and what I am would not be possible without the unyielding support of my nurturing family. Thank you to my parents, Tang Kim Poh and Tan Chuan Choo, for their selfless and unwavering love, confidence and encouragement.

Finally, my most heartfelt thanks go to Lin Tian: my best friend and biggest fan. My life will not be complete without her, much less doing a Ph.D. Her unconditional love and encouragement have accompanied me through the brightest of moments and have supported me through the darkest of times. She is a wonderful wife and a dream come true. Thank you Lin.

Beng Chiew (Adrian) Tang

Feb 22, 2018

To my wife, Lin Tian

for the wonderful life we built for ourselves

in New York and Singapore

1

Introduction

Security research is a curious field. Rarely do we encounter the need to take on such vastly disparate hats—attackers and defenders—in a field to do well. So intricately linked are these two roles that one arguably has to be well-versed in one to be proficient in the other; to be a good defender in securing systems, one has to think like an attacker. For years, the proverbial cat-and-mouse game between attack and defense has been fought where the lowest-hanging fruit lies – the relatively more exposed attack surface constituting the network, application and kernel layer of systems. Many prevalent and prolific attacks target these layers where users interact most with systems. It is thus no wonder that security engineering has traditionally taken a software-oriented approach, top-down from the system computing stack. Many defenses naturally adopt and maintain a purely software-only approach because compared to hardware, software is more flexible and enables faster time-to-deployment. This broadly represents software-based defenses adopting a *top-down* approach. On the other end of the spectrum is the body of hardware-based defenses that espouse the *bottom-up* approach by designing secure hardware from the get-go. Since hardware is relatively immutable[1], this approach adopts a longer-term view in designing security-oriented hardware primitives and developing hardware support for security. These software top-down and hardware bottom-up approaches encapsulate the two prevailing schools of philosophy towards engineering secure systems.

Despite progress made on the defense front, the increasing degree of heterogeneity (*e.g.* mo-

[1]Using microcode patches, modern commodity hardware possess some degree of mutability.

bile SoCs), ubiquity (*e.g.* IoT) and multi-layerness (*e.g.* disparate privilege and isolation execution modes) in systems have complicated security engineering. Apart from having more components to consider, system designers have to also examine how these components come together and anticipate ways in which hardware and software interact. Towards this end, this dissertation explores a third "*middle-out*" security engineering approach that can be more effectively tapped to further security – somewhere that lies between the bottom-up and top-down approaches. This requires the close examination of interfaces and the interaction between hardware and software.

This *middle-out* interface-oriented approach offers three benefits. One, it marries the strengths of hardware-based and software-based approaches to create defenses that are deployable in the immediate term. Software-only approaches have limits—performance penalty among others—that hardware is well-positioned to overcome. By repurposing existing hardware features not traditionally used for security, software-based defenses can be made more efficient and effective. Two, adopting an architectural view when thinking about defenses allows defender to uncovered untapped mechanisms for defense. Besides accelerating software security functions, hardware can enable new security mechanisms due to its extensive visibility of program execution. This holds much potential for new approaches to secure software systems. Third, it places emphasis on the cross-interaction between system components and motivates the security-aware design of both the system components and their interfaces.

It is thus the broad theme of my research to advance system security using hardware-software interfaces as a focal area to reformulate security measures and system designs. In this dissertation, I present three case studies to demonstrate the value of security engineering of hardware-software interfaces. Specifically, I motivate the redesign of existing security-oblivious energy management systems and their interfaces, repurpose hardware virtualization support to create an exploit prevention primitive, and adapt the traditional use of microarchitectural profiling in benchmarking and system optimization to detecting malware. This interface-oriented approach is practical and useful as the techniques demonstrated in this dissertation are applicable to all three key lifecycle stages of attacks.

1.1 Convergence of Attacks and Defenses at Interfaces

Characterizing the interworking of hardware and software, hardware-software interfaces offer a wealth of opportunities to both enhance as well as weaken systems. The focus on these interfaces is motivated notably by two prevailing trends in the industry and academia.

1.1.1 Hardware-Oriented Defenses

In the last two decades or so, to address threats to software security, increasing attention has been directed towards supporting security functions in hardware, both in academia and commercial products. Hardware vendors have introduced security-driven primitives directly in hardware, such as the Trusted Platform Module (TPM) to provide hardware-based root of trust, the hardware NX bit to support the enforcement of Data Execution Prevention (DEP), cryptographic accelerators [71] and random number generators [66]. Hardware-enforced isolation mechanisms on commodity systems such as ARM TrustZone [89] and Intel SGX [69] also brought on further research and development into Trusted Execution Environments. To thwart exploitation of memory safety violation bugs, vendors have begun to adapt software-only conceptions of Control Flow Integrity (CFI) [4], shadow call stack and Code-Pointer Integrity (CPI) [85] to hardware in the form of Intel Control Flow Enforcement (CET) [67] and Qualcomm ARMv8.3 Pointer Authentication [122]. There is also increasing support for hardware-based malware detection in platforms such as Qualcomm SnapDragon 820 [119]. This trend gradually marks the end of a long hiatus in hardware-oriented security research after robust research in the 1970's that enabled hardware support in the form of rings and virtualization support in hardware.

1.1.2 Hardware-Oriented Attacks

As most defenses place emphasis in securing the upper application and kernel layers of the system stack, it is not surprising to see a rising surge in recent years in attacks targeting the lower hardware, firmware and hypervisor layers of the system stack. Since security is a full-

system property, every layer of the computing stack is fair game to attackers. At the hardware layer, researchers have surfaced CPU bugs in chips [91] and reliability issues with DRAM memory [79]. The most prominent hardware-oriented attack in recent years is the "rowhammer" issue with DRAM. Researchers have since exploited "rowhammer" to demonstrate the dangers of such software-induced hardware-based transient bit-flips in practical scenarios ranging from browsers [56], virtualized environments [123], privilege escalation on Linux kernel [126] and from Android apps [156]. Besides exploiting hardware issues, attackers have also found and exploited bugs in esoteric firmware in Intel BIOS [173], Apple UEFI [152], networked printers [35] and ARM TEE [17]. As popularity of virtualized environments rises, attacks on hypervisors also emerge. Lower-privileged attack code running within virtualized environments can break out of isolation [171], cause the host systems to fail or even attack other virtual machines [111]. This trend clearly suggests that attackers are determined to attack *any* weak link, regardless of where in the computing stack the target is at.

1.2 Interface is the problem. Interface is the solution.

Interfaces allow users to interact with a system and also enable system components to interact with one another. So naturally, from the perspective of an attacker, interfaces constitute the attack surface of a system. All of the aforementioned hardware-oriented attacks involve attackers exploiting the system via some form of interface, however esoteric it may be. Interfaces are, thus, commonly associated with security risks or threats. In this work, I hope to offer a contrarian view of interfaces and discuss how approaching security oriented at these interfaces can lead to refreshing security insights and defenses.

1.2.1 Full-System Security

Security is a full-system property. As systems become more complex (*e.g.* increasingly heterogeneous hardware and software running with ever-increasing number of abstraction and privilege

levels), it becomes increasingly harder to think and review security in a piecemeal manner. This means that system design should improve in two ways. One, designers and architects of all system components, however minute, should include security as a design consideration. Two, the conventional approach to designing system components in silos should change. In other words, a coordinated full-system approach is likely required to further security moving forward.

To imbue urgency to the importance of such full-system design approach, this dissertation presents the first security review of an ubiquitous and indispensable system feature – energy management. Energy management is deliberately picked as a target for two reasons. First, despite the ubiquity of energy management mechanisms on commodity systems, security is rarely a consideration in the design of these mechanisms. Second, the design of energy management mechanisms demands design and optimizations that span across the entire computing stack. By demonstrating the fragility of such complex systems in terms of security, it is my hope that it gives credence to the need for full-system approach to designing for security.

1.2.2 Commodity Hardware Support for Security

A longer-term solution to security problems is to design hardware primitives securely from the get-go and build secure systems from ground up so to speak. However, due to market forces and short product cycles stressing on quick turnover, it sometimes takes a prolonged amount of time for security techniques to find their way into commodity systems. For example, it took the idea of transactional memory, originally proposed to mitigate data races [60], more than two decades to make it into commodity hardware in the form of Intel TSX [70]. The software-only conception of shadow stacks, first proposed by Abadi *et al.* [5] to enforce backward-edge control flow integrity, only recently (after more than one decade) began to make its way into commodity hardware recently in Intel CET [67]. How then can we protect existing COTS software? Relooking at the hardware features exposed to the software layers at the interfaces can offer a shorter-term development cycle for immediate protection.

Unconventional use of hardware features at the hardware-software interface layer offers a

reasonable balance between designing security into hardware (and waiting for the feature to pop up in commodity systems) and implementing software-based defenses (that may be slow and not taking enough advantage of hardware support). To show how commodity hardware features can be repurposed by exploiting their characteristics for the task at hand, this dissertation proposes the design and implementation of an exploit prevention primitive that leverages virtualization hardware support. This work harnesses the commodity virtualization hardware support in two ways. One, it uses hardware support to enable timely and transparent mediation of memory read operations. Two, it protects closed-source software and just-in-time compiled code against memory disclosure exploits. Since virtualization hardware support is available on commodity systems, this defense mechanism can be immediately made available to users.

1.2.3 Hardware-Software Interaction Matters

The most general abstraction for interfaces is that between hardware and software. Considering how hardware and software interact can open up new avenues for improving security. When software executes on hardware, their interaction can in fact be modeled by tracking real-time events at the underlying microarchitectural components. These events can include both microarchitectural ones such as cache misses and branch mispredictions, and architectural ones such as memory load and store operations. For years, hardware architects and software engineers have studied these profiled events to optimize both hardware and software design. Leveraging these microarchitectural profiles of software offers a new way to detect the execution of malware.

To demonstrate how microarchitectural interaction between hardware and software can be used to detect anomalous and malicious code execution, this dissertation develops a hardware performance counter-based framework that can efficiently profile these microarchitectural observables. This represents the first work to examine the feasibility and limits of using unsupervised learning on microarchitectural features from hardware performance counters to detect malware. More than about detecting malware, this work also shows that a nuanced look at hardware-software interaction actually matters, especially in thinking about security.

1.3 Thesis

1.3.1 Thesis Statement

The dissertation asserts that *security as a full-system property can be improved by examining and leveraging the interworking of hardware and software.*

1.3.2 Contributions

The dissertation makes the case for interface-oriented security engineering with the following contributions.

Security-aware full-system architecture design. The incidence of hardware-oriented and microarchitectural attacks has risen rapidly over the recent years. We are possibly at the inflexion point where attacks are increasingly shifting their focus to combining the manipulation of multiple independently-architected system components in novel ways to break system security. This dissertation brings to light the deficiency of the status quo in architecture design in face of such fast advancing attacks. We offer the first concrete security review of energy management, a crucial architectural component in systems. While the paper focuses on energy management, it highlights the value of offensive security research at large in both the architecture and systems community. By describing the challenges and how they can be overcome to pull off a successful attack, the dissertation quantifies the assumptions and resources needed for attackers to break existing and future designs. Exploring the plausibility of such new attack vectors and attacks is extremely valuable because they allow system designers—the defenders—to understand future attacks, and potentially prevent them from happening by creating appropriate defenses. As system designers work to invent and implement these protections, security researchers can complement these efforts by creating newer and exciting attacks on these protections. Architecture design should, therefore, adopt a full-system and security-aware approach to avoid introducing vulnerabilities early in its life cycle.

Hardware-assisted moving-target defense. Moving-target principles such as random-

ization that are employed in security architectures have proven inadequate in various ways in avoiding exploitation. This dissertation argues that such principles remain valuable as long as they are augmented with other forms of mechanisms to reduce or eliminate system knowledge exploitable by attackers. We show that destroying runtime information, *i.e.* introducing non-determinism into system states, is yet another principle to remove knowledge usable by attackers. In realizing a new exploit prevention primitive to demonstrate this principle, we ensure its immediate deployability and runtime performance by repurposing readily available hardware features, in our case hardware virtualization support. Novel software-based repurposing of commodity hardware features can serve to both facilitate new defenses and inform the future design of hardware security primitives.

Hardware-based anomaly sensors for attack detection. This dissertation explores the use of lower-level features that characterize the microarchitectural interaction between hardware and software to detect malicious code execution. We demonstrate that despite lacking higher-level semantic information, such previously untapped microarchitectural observables can be succinctly modeled and are efficient to audit. They are also harder to manipulate directly by attackers. This opens up the possibility of creating hardware-based sensors and incorporating them as standard elements in system designs. At a higher level, this work advocates the shift of defensive posture from a prevention/enforcement-heavy stance to one that assumes the inevitability of attacks and focuses on detecting and mitigating attacks.

1.3.3 Dissertation Roadmap

Chapter 2 outlines the general attack classes and defense principles in the context of the three primary lifecycle of security attacks. Chapter 3 presents the new CLKscrew attack vector enabled by security-oblivious energy management mechanisms. Chapter 4 introduces the destructive code read primitive implemented in a system dubbed Heisenbyte that thwarts dynamic code reuse attacks. Chapter 5 covers Hades, a malware detection system powered by hardware performance counters. Finally, the dissertation concludes in Chapter 6.

2

Background on Attacks and Defenses

2.1 Attack Model and Defense Principles

To discuss the value of hardware support in defenses, it is useful to understand the general classes of exploitation during different stages of an attack. In this chapter, we first detail the attack classes in our attack model. Then we highlight the key defense principles and goals encapsulated in the attack model. Shown in Figure 2.1, this juxtaposition of attack classes and defense principles enables us to systematize works in the context of the security principles they strive to secure, while discussing the tradeoffs of the use of hardware support.

Broadly speaking, an attack against a system comprises three main stages. First, the attacker has to induce some form of unintended system behavior to begin her attack. Once the attacker can get the system to function out of the intended specifications, she can exploit the system by violating one or more security properties. In the event that the she gains execution control of the system, she can finally conduct the attack proper on the system.

Stage ❶ - Unintended Behavior Unintended behavior typically results from a bug or logic error. An attacker can begin the attack by triggering a vulnerability that makes the system or program function beyond the intended design specifications. These unexpected behaviors manifest differently depending on where on the system stack the attack occurs. Therefore, we delineate the behaviors accordingly across the system stack to highlight the differing characteristics of the attack classes.

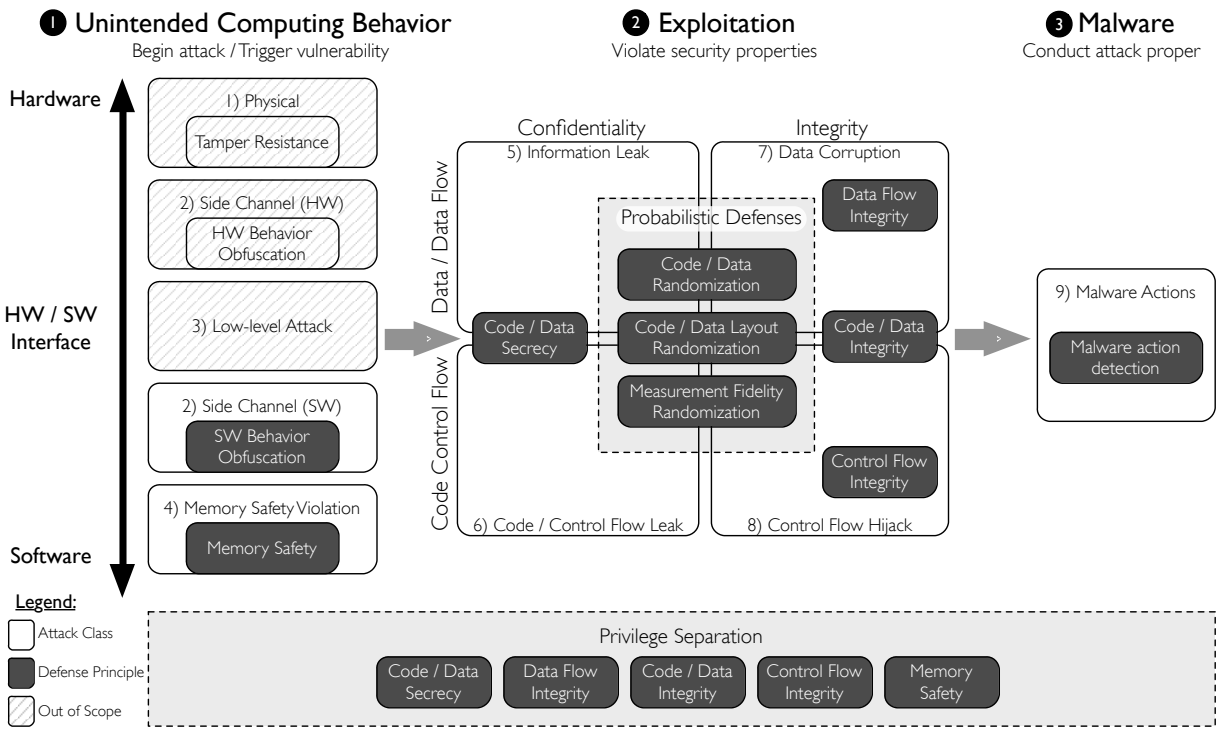

Figure 2.1: Attack classes juxtaposed with key defense principles and goals.

1) Physical Since hardware is the ultimate root of trust in a system, an attacker can conceivably break all security properties of a system once she gains physical access to the system. Some examples of physical attacks are cold boot attacks [55] to retrieve secrets from the memory, and power glitching attacks to bypass bootloader integrity checks.

- *Tamper resistance:* Defending against this class of attacks requires tamper-resistant secure hardware design or enforcing physical security to the hardware. Since our paper focuses on the role of hardware support in defenses against software-originated attacks, we do not consider these attacks that assume physical access to the target systems.

2) Side Channel Attack Side channel attacks can be used to infer secrets from systems using observable runtime side-effects of program execution. These side-effects can be manifestation of either physical aspects of the systems such as heat, radio frequency emanations or vibration, or interaction between the hardware and software such as cache timing differences. We term the

former hardware-based side channel attacks and the latter software-based ones. Since the former requires physical access to probe the observations physically, we consider them out of scope. Our attack model includes the latter class since it can be conducted using software.

- *HW Behavior Obfuscation:* Since side channel attacks rely on the unintended correlation between the confidential information and some form of observable leaked emissions, the main defense goal is to eliminate or obfuscate this implicit relationship at the hardware level. An example is Oblivious RAM (ORAM). We do not consider defenses that try to achieve this defense property.
- *SW Behavior Obfuscation:* Changing or obfuscating parts of program behavior can eliminate or reduce the link between any observables and the information to be inferred. This can involve control flow obfuscation [29] or executing diversified software [34].

3) Low-level Attack Low-level attacks are software-originated attacks that leverage hardware-based vulnerabilities [48]. They can involve firmware (*i.e.* hardware-specific software executed close to the hardware layer) such as UEFI [172] and BIOS [26], or hardware components such as DMA [141], processor erratas [166], or DRAM [80].

- Since these attacks involve hardware, the most effective defense is to fix the problem in hardware, or patching the firmware or microcode using updates. These low-level attacks and defenses are out of scope for our paper.

4) Memory Safety Violation Most prevalent system attacks begin with a violation of memory safety in software developed with type-unsafe languages like C/C++. Frequently referred to as memory corruption bugs [143], memory safety vulnerabilities allow attackers to achieve arbitrary memory read and write capabilities beyond the target program's intended specifications, and thus form a key enabler in an attack. Examples of attacks violating the spatial form of memory safety are buffer overflow attacks; dangling pointer attacks are exemplary cases of the temporal version of memory safety violations.

- *Memory safety:* Achieving memory safety in compiled programs is making it impossible to induce program execution to read or write to memory locations other than those intended by the developers. Ensuring memory safety eliminates memory corruption vulnerabilities. It includes preventing reading of uninitialized memory, accessing freed memory and performing illegal operations on the heap memory like double frees.

Stage ❷ - Exploitation This stage occurs after the attacker is able to make the system function outside the intended specifications. While the earlier stage 1 can be viewed as an attack fulfilling a pre-requisite of initial control, this stage 2 characterizes the various ways security properties of a system are compromised. This characterization can be viewed across two main dimensions. On one, we focus on two main security properties, namely confidentiality and integrity, of a system. On the other, we characterize the compromise of security properties based on what form of information they pertain to.

5) Information Leak This class of attack breaks the confidentiality of data and data flow, and reveals to the attacker sensitive information like cryptographic keys. This typically follows from a memory safety violation that gives the attacker arbitrary memory read capability. A recent example of an information leak attack is one that exploits the Heartbleed vulnerability in OpenSSL to leak sensitive memory contents on systems [92]. Apart from direct information leak, sensitive data can also be disclosed via side channel attacks that infer the data without directly reading the memory.

6) Code / Control Flow Leak The secrecy of code can be compromised with arbitrary memory read capabilities achieved from memory disclosure vulnerabilities. An attacker can discover code instructions in executable memory by directly reading code memory [133, 142] or indirectly inferring code layout using code pointers discovered from data memory pages [36, 33]. With time and fault analysis side channels, an attacker can also learn information about the code without arbitrary memory read capabilities [127].

- *Code / Data secrecy:* To deal with unintended leakage of code or data, the confidential-

ity of code and data in memory can be protected by means of encryption or obfuscation, or by preventing read or write operations to the memory (such as execute-only memory (XOM) [149]).

7) Data Corruption After achieving arbitrary memory write capability from stage 1, an attacker can compromise the integrity of data in the memory. Data corruption attacks are exemplified by non-control-data attacks [30] that overwrite security-critical application-specific data such as user identity data or configuration data, without subverting the intended control flow on systems.

- *Data / Data flow integrity:* To prevent unintended corruption of security-sensitive data, a key defense strategy is to enforce data flow integrity at runtime. Data flow integrity requires deriving the static data flow graph of a program and ensuring that any data flow at runtime complies with the static data flow graph, possibly with the use of program instrumentation [28].

8) Control Flow Hijack The most prevalent form of attacks, control flow hijack breaches the integrity of original control flow of a targeted program by subverting the original program instruction pointer to the attacker's choosing. The hijacking of the original program control flow necessarily violates the intended static and dynamic control flow graph of the program.

- *Control flow integrity:* Analogous to data flow integrity, control flow integrity is the security property that guarantees all runtime control flow transfers are the ones intended by the original program [5].
- *Code integrity:* Ensuring the integrity of code and data in memory requiring tamper-proofing code and data from any unauthorized write operations.

Stage ❸ - Malware Proper

9) Malware Actions The final step of an attack is performing the attacker's malicious goals. This stage typically constitutes an injected or code-reuse shellcode, as well as a full-fledged malicious program payload.

- *Malware action detection:* Once the actual attack proper is underway in an attack, these malicious actions manifest in various forms across the system stack (such as power consumption patterns [77], microarchitectural side-effects [37], system calls [62] and functions [110]). The detection of these malware actions can take place during the execution of the shellcode [169] used during the actual exploitation of the bug or the purpose-driven payload after the attacker has full control over the system.

2.1.1 Generic Defense Principles

While we highlight *localized* defense principles that target specific attack classes in the preceding section, here, we describe *generic* defense approaches that target multiple attack classes at the same time.

Probabilistic Defenses Probabilistic defenses work by making a system less deterministic by means of randomization.

- *Code / Data randomization:* Instruction Set Randomization (ISR) [75] thwarts code injection attacks by changing the instruction set on a per-process basis, so that any injected foreign code without the knowledge of the randomized instruction set will fail to function as intended.
- *Code / Data layout randomization:* Randomizing the locations of code and data in memory makes it harder for an attacker to identify usable code or data needed for her exploits. The goal of memory layout randomization is to eliminate any a priori information gained from analyzing the programs before the attack. A widely deployed example of such layout randomization is Address Space Layout Randomization (ASLR) [148].

- *Measurement fidelity randomization* Side channel attacks frequently require making precise measurements of the observable effects of program execution. One line of defense involves reducing the fidelity of any measurement facilities so that the added noise decreases to signal-to-noise ratio in the side channel attack. For example, to prevent timing attacks, random delays can be introduced as noise in systems to make it difficult for an attacker to infer any secret information [82].

Privilege Separation Privilege is defined as the delegation of authority over a system resources, or more informally the permission to perform some system action. In the most general sense, achieving privilege separation implies that all components of a system are restricted to only resources they are allowed to read, write or execute access. This is a generic defense principle that can mitigate several attack classes at the same time.

There are three main approaches to monitor and enforce privilege separation, differing in the way they abstract privileges. Isolation-based techniques rely on "rings" of hierarchies or some form of sandboxing. Capability-based techniques leverage the possession of "tokens" – an access control construct, to enforce permissions. Using tagging, information flow tracking techniques track the provenance of data and mediate access.

3

CLKscrew: Motivating Security-Aware Energy Management

Aggressive commodity hardware-software energy management mechanisms open up a new class of software-induced fault attack vector.

Security-oblivious energy management mechanisms can be exploited to compromise hardware-enforced security isolation.

The need for power- and energy-efficient computing has resulted in aggressive cooperative hardware-software energy management mechanisms on modern commodity devices. Most systems today, for example, allow software to control the frequency and voltage of the underlying hardware at a very fine granularity to extend battery life. Despite their benefits, these software-exposed energy management mechanisms pose grave security implications that have not been studied before.

In this chapter, we present a study of contemporary energy management designs and show how a series of well-thought-out, nevertheless security-oblivious, design decisions can create security vulnerabilities. To demonstrate that these risks are real and practical, we formulate the CLKscrew attack, a new class of fault attacks that exploit the security-obliviousness of energy management mechanisms to break security. This new attack vector is dangerous because it makes fault attacks more accessible to attackers since the attacks can now be conducted without the need for physical access to the devices or fault injection equipment. We demonstrate CLKscrew [146] on commodity ARM/Android devices. We show that a malicious kernel driver

(1) can extract secret cryptographic keys from Trustzone, and (2) can escalate its privileges by loading self-signed code into Trustzone. As the first work to show the security ramifications of energy management mechanisms, we hope to motivate security-aware energy management in existing and future designs, both in industry and academia.

3.1 Overview

The growing cost of powering and cooling systems has made energy management an essential feature of most commodity devices today. Energy management is crucial for reducing cost, increasing battery life, and improving portability for systems, especially mobile devices. Designing effective energy management solutions, however, is a complex task that demands cross-stack design and optimizations: Hardware designers, system architects, and kernel and application developers have to coordinate their efforts across the entire hardware/software system stack to minimize energy consumption and maximize performance. Take as an example, Dynamic Voltage and Frequency Scaling (DVFS) [109], a ubiquitous energy management technique that saves energy by regulating the frequency and voltage of the processor cores according to runtime computing demands. To support DVFS, at the hardware level, vendors have to design the underlying frequency and voltage regulators to be portable across a wide range of devices while ensuring cost efficiency. At the software level, kernel developers need to track and match program demands to operating frequency and voltage settings to minimize energy consumption for those demands. Thus, to maximize the utility of DVFS, hardware and software function cooperatively and at very fine granularities.

Despite the ubiquity of energy management mechanisms on commodity systems, security is rarely a consideration in the design of these mechanisms. In the absence of known attacks, given the complexity of hardware-software interoperability needs and the pressure of cost and time-to-market concerns, the designers of these mechanisms have not given much attention to the security aspects of these mechanisms; they have been focused on optimizing the functional

aspects of energy management. These combination of factors along with the pervasiveness of these mechanisms makes energy management mechanisms a potential source of security vulnerabilities and an attractive target for attackers.

In this work, we present the first security review of a widely-deployed energy management technique, DVFS. Based on careful examination of the interfaces between hardware regulators and software drivers, we uncover a new class of exploitation vector, which we term as CLKscrew. In essence, a CLKscrew attack exploits unfettered software access to energy management hardware to push the operating limits of processors to the point of inducing faulty computations. This is dangerous when these faults can be induced from lower privileged software across hardware-enforced boundaries, where security sensitive computations are hosted.

We demonstrate that CLKscrew can be conducted using no more than the software control of energy management hardware regulators in the target devices. CLKscrew is more powerful than traditional physical fault attacks [20] for several reasons. Firstly, unlike physical fault attacks, CLKscrew enables fault attacks to be conducted purely from software. Remote exploitation with CLKscrew becomes possible without the need for physical access to target devices. Secondly, many equipment-related barriers, such as the need for soldering and complex equipment, to achieve physical fault attacks are removed. Lastly, since physical attacks have been known for some time, several defenses, such as special hardened epoxy and circuit chips that are hard to access, have been designed to thwart such attacks. Extensive hardware reverse engineering may be needed to determine physical pins on the devices to connect the fault injection circuits [105]. CLKscrew sidesteps all these risks of destroying the target devices permanently.

To highlight the practical security impact of our attack, we implement the CLKscrew attack on a commodity ARMv7[1] phone, Nexus 6. With only publicly available knowledge of the Nexus 6 device, we identify the operating limits of the frequency and voltage hardware mechanisms. We then devise software to enable the hardware to operate beyond the vendor-recommended limits. Our attack requires no further access beyond a malicious kernel driver. We show how the

[1]As of Sep 2016, ARMv7 devices capture over 86% of the worldwide market share of mobile phones [101].

CLKscrew attack can subvert the hardware-enforced isolation in ARM Trustzone in two attack scenarios: (1) extracting secret AES keys embedded within Trustzone and (2) loading self-signed code into Trustzone. We note that the root cause for CLKscrew is neither a hardware nor a software bug: CLKscrew is achievable due to the fundamental design of energy management mechanisms.

We have responsibly disclosed the vulnerabilities identified in this work to the relevant SoC and device vendors. They have been very receptive to the disclosure. Besides acknowledging the highlighted issues, they were able to reproduce the reported fault on their internal test device within three weeks of the disclosure. They are working towards mitigations.

In summary, we make the following contributions in this chapter:

1. We expose the dangers of designing energy management mechanisms without security in mind by introducing the concept of the CLKscrew attack. Aggressive energy-aware computing mechanisms can be exploited to influence isolated computing.
2. We present the CLKscrew attack to demonstrate a new class of energy management-based exploitation vector that exploits software-exposed frequency and voltage hardware regulators to subvert trusted computation.
3. We introduce a methodology for examining and demonstrating the feasibility of the CLKscrew attack against commodity ARM devices running a full OS such as Android.
4. We demonstrate that the CLKscrew attack can be used to break ARM Trustzone by inferring secret cryptographic keys and loading self-signed applications on a commodity phone.

3.2 Background

In this section, we first an overview of the design of energy management systems. Next we describe DVFS and how it relates to saving energy. We then detail key classes of supporting hardware regulators and their software-exposed interfaces.

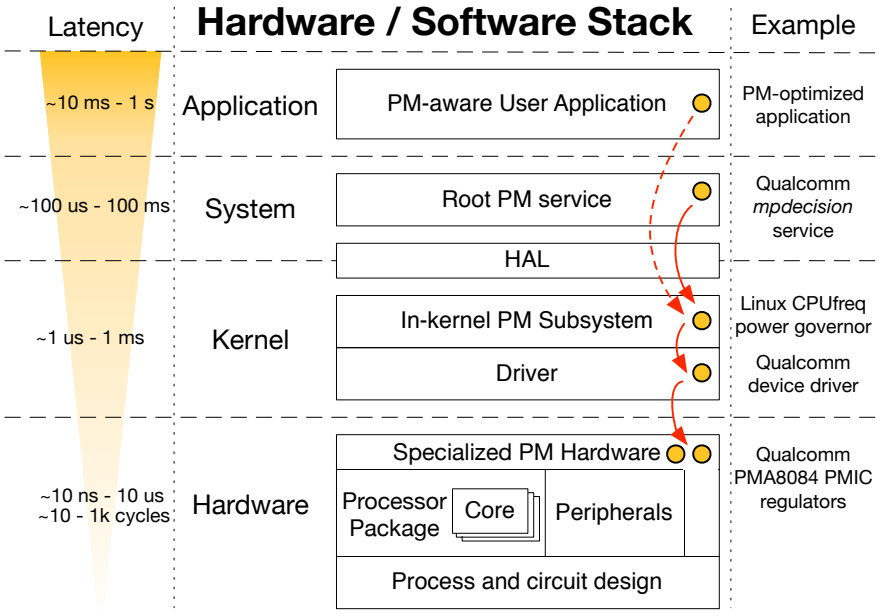

Figure 3.1: Contemporary energy management system designs span across multiple layers in the computing stack.

3.2.1 Energy Management Systems

As part of the attacks, we examine the implementations of contemporary energy management systems. Current energy management systems employ cross-stack design and optimizations in practice. Figure 3.1 shows a typical energy management system design and its various components across the system stack. The energy management system spans from the application layer all the way to the hardware layer. The figure also shows the layer-specific transition latencies and examples of the corresponding energy management components in each layer. Two requirements underpin the prevalence of such layered designs. One, optimizing system performance with respect to energy consumption requires accurate feedback on the runtime system workload. Components in specific system layers may be more equipped to provide certain feedback information. For example, instantaneous system utilization levels are readily available to the OS kernel layer. As such, the Linux CPUfreq power governor is well-positioned at that layer to initiate runtime changes to the operating voltage and frequency based on these whole-system measures. Two, components in the higher layers incur higher transition latencies due to cross-

layer invocation over-head. Since latencies from the lower layers propagate and accumulate into the upper layers, the hardware regulators are designed to be most responsive among their software counterparts. Although the upper-layer components are slower, their timing requirements are less stringent.

In the following sections, we will review the energy management systems specific to the ARMv7 SoCs. We argue that the energy management design issues are by no means specific to any given architecture. For reference, we also provide a deep dive into the energy and power management controls on the Intel platforms in Appendix A.2.

3.2.2 Dynamic Voltage & Frequency Scaling

DVFS is an energy management technique that trades off processing speed for energy savings. Since its debut in 1994 [167], DVFS has become ubiquitous in almost all commodity devices. DVFS works by regulating two important runtime knobs that govern the amount of energy consumed in a system – frequency and voltage.

To see how managing frequency and voltage can save energy, it is useful to understand how energy consumption is affected by these two knobs. The amount of energy[2] consumed in a system is the product of power and time, since it refers to the total amount of resources utilized by a system to complete a task over time. Power[3], an important determinant of energy consumption, is directly proportional to the product of operating frequency and voltage. Consequently, to save energy, many energy management techniques focus on efficiently optimizing both frequency and voltage.

DVFS regulates frequency and voltage according to runtime task demands. As these demands can vary drastically and quickly, DVFS needs to be able to track these demands and effect the frequency and voltage adjustments in a timely manner. To achieve this, DVFS requires components

[2]Formally, the total amount of energy consumed, E_T, is the integral of instantaneous dynamic power, P_t over time T: $E_T = \int_0^T P_t \, \mathrm{d}t$.

[3]In a system with a fixed capacitative load, at any time t, the instantaneous dynamic power is proportional to both the voltage, V_t and the frequency F_t as follows: $P_t \propto V_t^2 \times F_t$.

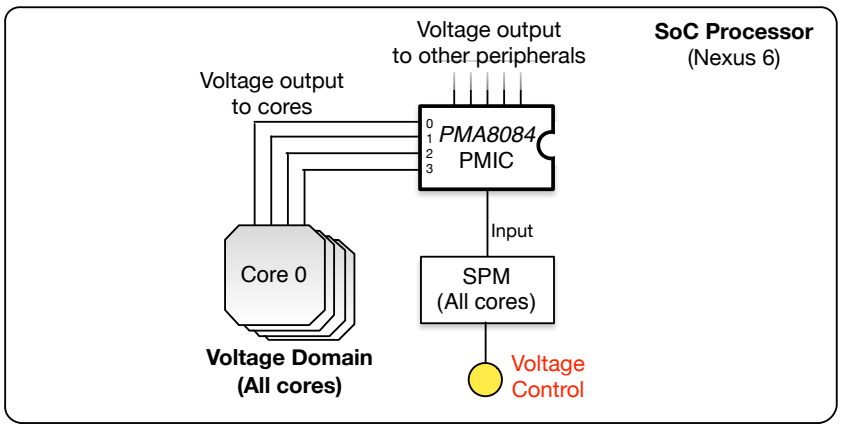

Figure 3.2: Shared voltage regulator for all *Krait* cores.

across layers in the system stack. The three primary components are (1) the voltage/frequency hardware regulators, (2) vendor-specific regulator driver, and (3) OS-level *CPUfreq* power governor [106]. The combined need for accurate layer-specific feedback and low voltage/frequency scaling latencies drives the prevalence of unfettered and software-level access to the frequency and voltage hardware regulators.

3.2.3 Hardware Support for DVFS

Voltage Regulators. Voltage regulators supply power to various components on devices, by reducing the voltage from either the battery or external power supply to a range of smaller voltages for both the cores and the peripherals within the device. To support features, such as camera and sensors that are sourced from different vendors and hence operating at different voltages, numerous voltage regulators are needed on devices. These regulators are integrated within a specialized circuit called Power Management Integrated Circuit (PMIC) [131].

Power to the application cores is typically supplied by the step-down regulators within the PMIC on the System-on-Chip (SoC) processor. As an example, Figure 3.2 shows the PMIC that regulates the shared voltage supply to all the application cores (*a.k.a. Krait* cores) on the Nexus 6 device. The PMIC does not directly expose software interfaces for controlling the voltage supply to the cores. Instead, the core voltages are indirectly managed by a power management subsys-

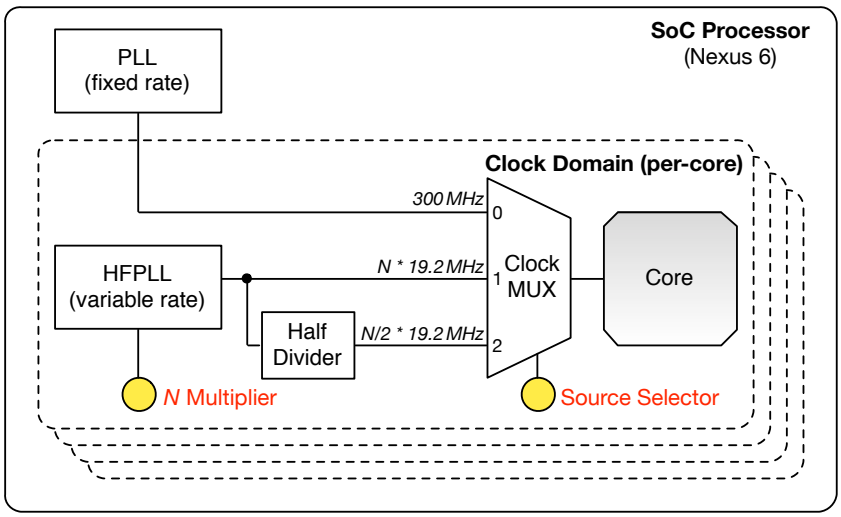

Figure 3.3: Separate clock sources for each *Krait* core.

tem, called the Subsystem Power Manager (SPM) [93]. The SPM is a hardware block that maintains a set of control registers which, when configured, interfaces with the PMIC to effect voltage changes. Privileged software like a kernel driver can use these memory-mapped control registers to direct voltage changes. We highlight these software-exposed controls as yellow-shaded circles in Figure 3.2.

Frequency PLL-based Regulators. The operating frequency of application cores is derived from the frequency of the clock signal driving the underlying digital logic circuits. The frequency regulator contains a Phase Lock Loop (PLL) circuit, a frequency synthesizer built into modern processors to generate a synchronous clock signal for digital components. The PLL circuit generates an output clock signal of adjustable frequency, by receiving a fixed-rate reference clock (typically from a crystal oscillator) and raising it based on an adjustable multiplier ratio. The output clock frequency can then be controlled by changing this PLL multiplier.

For example, each core on the Nexus 6 has a dedicated clock domain. As such, the operating frequency of each core can be individually controlled. Each core can operate on three possible clock sources. In Figure 3.3, we illustrate the clock sources as well as the controls (shaded in yellow) exposed to the software from the hardware regulators. A multiplexer (*MUX*) is used to

select amongst the three clock sources, namely (1) a PLL supplying a fixed-rate 300-MHz clock signal, (2) a High-Frequency PLL (HFPLL) supplying a clock signal of variable frequency based on a *N multiplier*, and (3) the same HFPLL supplying half the clock signal via a frequency divider for finer-grained control over the output frequency.

As shown in Figure 3.3, the variable output frequency of the HFPLL is derived from a base frequency of 19.2MHz and can be controlled by configuring the *N multiplier*. For instance, to achieve the highest core operating frequency of 2.65GHz advertised by the vendor, one needs to configure the *N multiplier* to 138 and the *Source Selector* to 1 to select the use of the full HFPLL. Similar to changing voltage, privileged software can initiate per-core frequency changes by writing to software-exposed memory-mapped PLL registers, shown in Figure 3.3.

3.2.4 Software Support for DVFS

On top of the hardware regulators, additional software support is needed to facilitate DVFS. Studying these supporting software components for DVFS enables us to better understand the interfaces provided by the hardware regulators. Software support for DVFS comprises two key components, namely vendor-specific regulator drivers and OS-level power management services.

Besides being responsible for controlling the hardware regulators, the vendor-provided PMIC drivers [117, 118] also provide a convenient means for mechanisms in the upper layers of the stack, such as the Linux *CPUfreq* power governor [106] to dynamically direct the voltage and frequency scaling. DVFS requires real-time feedback on the system workload profile to guide the optimization of performance with respect to power dissipation. This feedback may rely on layer-specific information that may only be efficiently accessible from certain system layers. For example, instantaneous system utilization levels are readily available to the OS kernel layer. As such, the Linux *CPUfreq* power governor is well-positioned at that layer to initiate runtime changes to the operating voltage and frequency based on these whole-system measures. This also provides some intuition as to why DVFS cannot be implemented entirely in hardware.

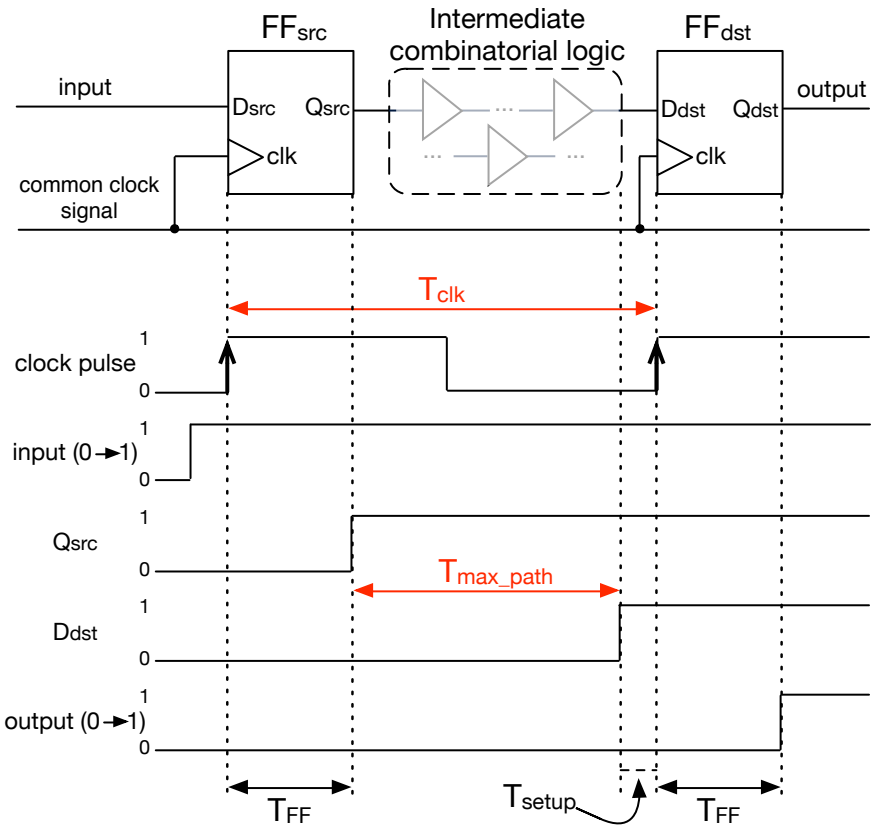

Figure 3.4: Timing constraint for error-free data propagation from input Q_{src} to output D_{dst} for entire circuit.

3.3 Achieving the First CLKscrew Fault

In this section, we first briefly describe why erroneous computation occurs when frequency and voltage are stretched beyond the operating limits of digital circuits. Next, we outline challenges in conducting a non-physical probabilistic fault injection attack induced from software. Finally, we characterize the operating limits of regulators and detail the steps to achieving the first CLKscrew fault on a real device.

3.3.1 How Timing Faults Occur

To appreciate why unfettered access to hardware regulators is dangerous, it is necessary to understand in general why over-extending frequency (*a.k.a.* overclocking) or under-supplying voltage (*a.k.a.* undervolting) can cause unintended behavior in digital circuits.

Synchronous digital circuits are made up of memory elements called flip-flops (FF). These flip-flops store stateful data for digital computation. A typical flip-flop has an input D, and an output Q, and only changes the output to the value of the input upon the receipt of the rising edge of the clock (CLK) signal. In Figure 3.4, we show two flip-flops, FF_{src} and FF_{dst} sharing a common clock signal and some intermediate combinatorial logic elements. These back-to-back flip-flops are building blocks for pipelines, which are pervasive throughout digital chips and are used to achieve higher performance.

Circuit timing constraint. For a single flip-flop to properly propagate the input to the output locally, there are three key timing sub-constraints. (1) The incoming data signal has to be held stable for T_{setup} during the receipt of the clock signal, and (2) the input signal has to be held stable for T_{FF} within the flip-flop after the clock signal arrives. (3) It also takes a minimum of $T_{\text{max\_path}}$ for the output Q_{src} of FF_{src} to propagate to the input D_{dst} of FF_{dst}. For the overall circuit to propagate input $D_{\text{src}} \rightarrow$ output Q_{dst}, the minimum required clock cycle period[4], T_{clk}, is bounded by the following timing constraint (3.1) for some microarchitectural constant K:

$$T_{\text{clk}} \geq T_{\text{FF}} + T_{\text{max\_path}} + T_{\text{setup}} + K \tag{3.1}$$

Violation of timing constraint. When the timing constraint is violated during two consecutive rising edges of the clock signal, the output from the source flip-flop FF_{src} fails to latch properly in time as the input at the destination flip-flop FF_{dst}. As such, the FF_{dst} continues to operate with stale data. There are two situations where this timing constraint can be violated, namely (a) overclocking to reduce T_{clk} and (b) undervolting to increase the overall circuit propagation time, thereby increasing $T_{\text{max\_path}}$. Figure 3.5 illustrates how the output results in an unintended erroneous value of 0 due to overclocking.

For comparison, we show an example of a bit-level fault due to undervolting in Figure 3.6. Note that in this case, the fault occurs because the propagation time increases beyond an achiev-

[4] T_{clk} is simply the reciprocal of the clock frequency.

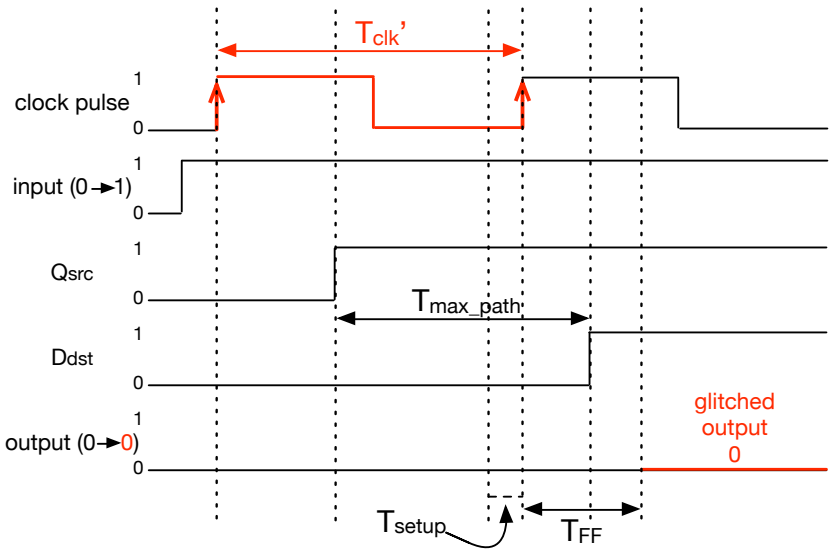

Figure 3.5: Bit-level fault due to overclocking: Reducing clock period $T_{\text{clk}} \rightarrow T'_{\text{clk}}$ results in a bit-flip in output $1 \rightarrow 0$.

able deadline.

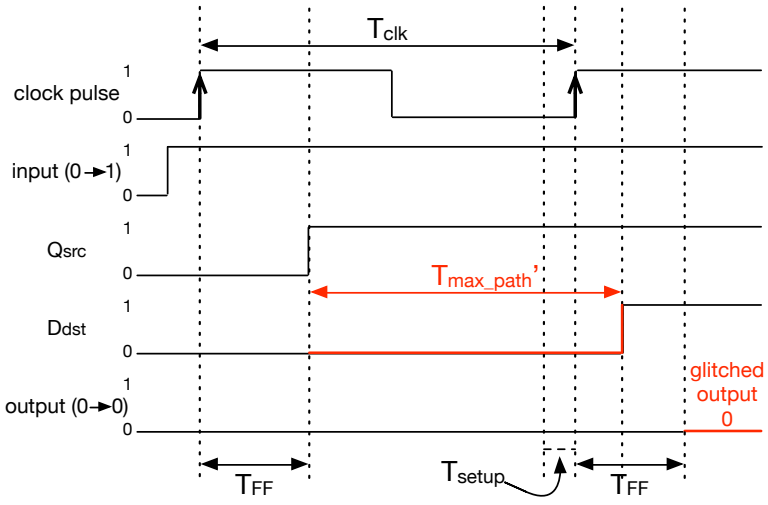

Figure 3.6: Glitch due to undervolting: Increasing propagation time of the critical path between the two consecutive flip-flops, $T_{\text{max\_path}} \rightarrow T'_{\text{max\_path}}$ results in a bit-flip in output $1 \rightarrow 0$.

3.3.2 Challenges of CLKscrew Attacks

Mounting a fault attack purely from software on a real-world commodity device using its internal voltage/frequency hardware regulators has numerous difficulties. These challenges are non-existent or vastly different from those in traditional physical fault attacks (that commonly use laser, heat and radiation).

Regulator operating limits. Overclocking or undervolting attacks require the hardware to be configured *far beyond* its vendor-suggested operating range. Do the operating limits of the regulators enable us to effect such attacks in the first place? We show that this is feasible in § 3.3.3.

Self-containment within same device. Since the attack code performing the fault injection and the victim code to be faulted both reside on the same device, the fault attack must be conducted in a manner that does not affect the execution of the attacking code. We present techniques to overcome this in § 3.3.4.

Noisy complex OS environment. On a full-fledged OS with interrupts, we need to inject a fault into the target code without causing too much perturbation to non-targeted code. We address this in § 3.3.4.

Precise timing. To attack the victim code, we need to be relatively precise in *when* the fault is induced. Using two attack scenarios that require vastly different degrees of timing precision in § 3.4 and § 3.5, we demonstrate how the timing of the fault can be fine-tuned using a range of execution profiling techniques.

Fine-grained timing resolution. The fault needs to be *transient* enough to occur during the intended region of victim code execution. We may need the ability to target a specific range of code execution that takes orders of magnitude fewer clock cycles within an entire operation. For example, in the attack scenario described in Section § 3.5.3, we seek to inject a fault into a memory-specific operation that takes roughly 65,000 clock cycles within an entire RSA certificate chain verification operation spanning over 1.1 billion cycles.

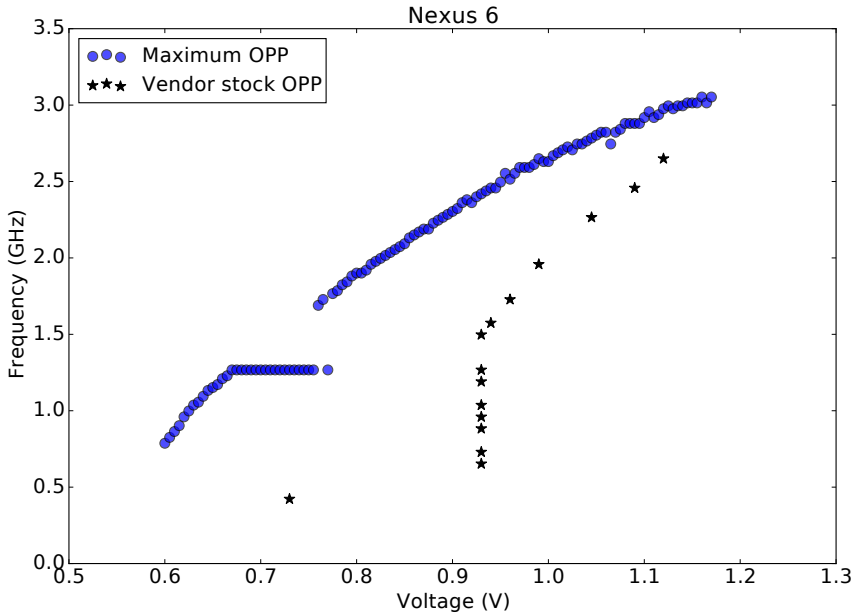

Figure 3.7: Vendor-stipulated voltage/frequency Operating Performance Points (OPPs) *vs.* maximum OPPs achieved before computation fails.

3.3.3 Characterization of Regulator Limits

In this section, we study the capabilities and limits of the built-in hardware regulators, focusing on the Nexus 6 phone. According to documentation from the vendor, Nexus 6 features a 2.7GHz quad-core SoC processor. On this device, DVFS is configured to operate only in one of 15 possible discrete[5] Operating Performance Points (OPPs) at any one time, typically by a DVFS OS-level service. Each OPP represents a state that the device can be in with a voltage and frequency pair. These OPPs are readily available from the vendor-specific definition file, `apq8084.dtsi`, from the kernel source code [104].

To verify that the OPPs are as advertised, we need measurement readings of the operating voltage and frequency. By enabling the *debugfs* feature for the regulators, we can get per-core voltage[6] and frequency[7] measurements. We verify that the *debugfs* measurement readings indeed

[5]A limited number of discrete OPPs, instead of a range of continuous voltage/frequency values, is used so that the time taken to validate the configured OPPs at runtime is minimized.

[6]`/d/regulator/krait`*X*`/voltage`

[7]`/d/clk/krait`*X*`_clk/measure`

match the voltage and frequency pairs stipulated by each OPP. We plot these vendor-provided OPP measurements as black-star symbols in Figure 3.7.

No safeguard limits in hardware. Using the software-exposed controls described in § 3.2.3, while maintaining a low base frequency of 300MHz, we configure the voltage regulator to probe for the range during which the device remains functional. We find that when the device is set to any voltage outside the range 0.6V to 1.17V, it either reboots or freezes. We refer to the phone as being *unstable* when these behaviors are observed. Then, stepping through 5mV within the voltage range, for each operating voltage, we increase the clock frequency until the phone becomes *unstable*. We plot each of these maximum frequency and voltage pair (as shaded circles) together with the vendor-stipulated OPPs (as shaded stars) in Figure 3.7. It is evident that the hardware regulators can be configured past the vendor-recommended limits. This unfettered access to the regulators offers a powerful primitive to induce a software-based fault.

> Attack Enabler (General) #1: There are no safeguard limits in the hardware regulators to restrict the range of frequencies and voltages that can be configured.

Large degree of freedom for attacker. Figure 3.7 illustrates the degree of freedom an attacker has in choosing the OPPs that have the potential to induce faults. The maximum frequency and voltage pairs (*i.e.* shaded circles in Figure 3.7) form an almost continuous upward-sloping curve. It is noteworthy that all frequency and voltage OPPs *above* this curve represent potential candidate values of frequency and voltage that an attacker can use to induce a fault.

This "shaded circles" curve is instructive in two ways. First, from the attacker's perspective, the upward-sloping nature of the curve means that reducing the operating voltage simultaneously lowers the minimum required frequency needed to induce a fault in an attack. For example, suppose an attacker wants to perform an overclocking attack, but the frequency value she needs to achieve the fault is beyond the physical limit of the frequency regulator. With the help of this frequency/voltage characteristic, she can then possibly reduce the operating voltage to the extent where the overclocking frequency required is within the physical limit of the regulator.

Attack Enabler (General) #2: Reducing the operating voltage lowers the minimum required frequency needed to induce faults.

Secondly, from the defender's perspective, the large range of instability-inducing OPPs above the curve suggests that limits of both frequency and voltage, if any, must be enforced *in tandem* to be effective. Combination of frequency and voltage values, while individually valid, may still cause unstable conditions when used together.

Prevalence of Regulators. The lack of safeguard limits within the regulators is not specific to Nexus 6. We observe similar behaviors in devices from other vendors. For example, the frequency/voltage regulators in the Nexus 6P and Pixel phones can also be configured beyond their vendor-stipulated limits to the extent of seeing instability on the devices. We show the comparison of the vendor-recommended and the actual observed OPPs of these devices in Figures 3.8 and 3.9.

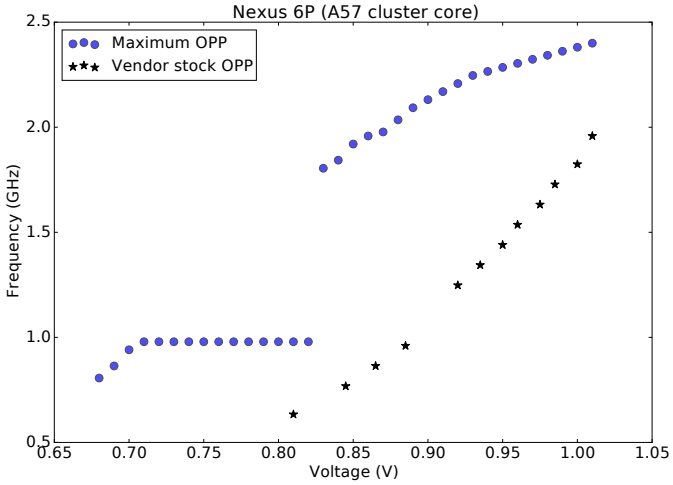

Figure 3.8: Vendor-stipulated vs maximum voltage/frequency OPPs for Nexus 6P.

3.3.4 Containing the Fault within a Core

The goal of our fault injection attack is to induce errors to specific victim code execution. The challenge is doing so without self-faulting the attack code and accidentally attacking other non-targeted code.

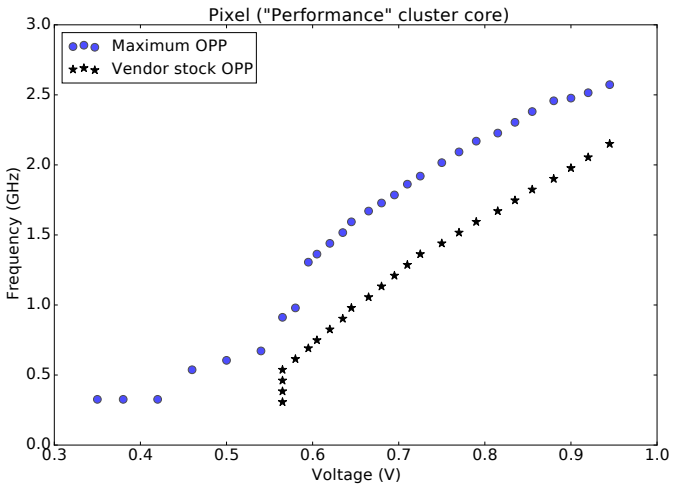

Figure 3.9: Vendor-stipulated vs maximum voltage/frequency OPPs for Pixel.

We create a custom kernel driver to launch separate threads for the attack and victim code and to pin each of them to separate cores. Pinning the attack and victim code in separate cores automatically allows each of them to execute in different frequency domains. This core pinning strategy is possible due to the deployment of increasingly heterogeneous processors like the ARM big.LITTLE [1] architecture, and emerging technologies such as Intel PCPS [72] and Qualcomm aSMP [121]. The prevailing industry trend of designing finer-grained energy management favors the use of separate frequency and voltage domains across different cores. In particular, the Nexus 6 SoC that we use in our attack is based on a variant of the aSMP architecture. With core pinning, the attack code can thus manipulate the frequency of the core that the victim code executes on, without affecting that of the core the attack code is running on. In addition to core pinning, we also disable interrupts during the entire victim code execution to ensure that no context switch occurs for that core. These two measures ensure that our fault injection effects are contained within the core that the target victim code is running on.

Attack Enabler (General) #3: The deployment of cores in different voltage/frequency domains isolates the effects of cross-core fault attack.

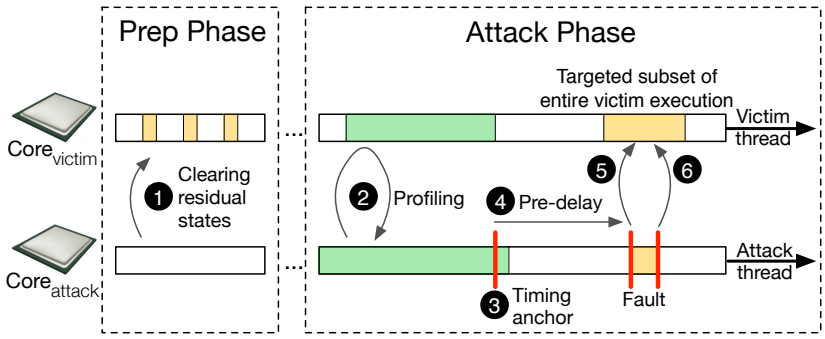

Figure 3.10: Overview of CLKscrew fault injection setup.

3.3.5 CLKscrew Attack Steps

Parameter	Description
F_{volt}	Base operating voltage
F_{pdelay}	Number of loops to delay/wait before the fault
$F_{\text{freq\_hi}}$	Target value to raise the frequency *to* for the fault
$F_{\text{freq\_lo}}$	Base value to raise the frequency *from* for the fault
F_{dur}	Duration of the fault in terms of number of loops

Table 3.1: CLKscrew fault injection parameters.

The CLKscrew attack is implemented with a kernel driver to attack code that is executing at a higher privilege than the kernel. Examples of such victim code are applications running within isolation technologies such as ARM Trustzone [2] and Intel SGX [7]. In Figure 3.10, we illustrate the key attack steps within the thread execution of the attack and victim code. The goal of the CLKscrew attack is to induce a fault in a subset of an entire victim thread execution.

(1) Clearing residual states. Before we attack the victim code, we want to ensure that there are no microarchitectural residual states remaining from prior executions. Since we are using a cache-based profiling technique in the next step, we want to make sure that the caches do not have any residual data from non-victim code before each fault injection attempt. To do so, we invoke both the victim and attack threads in the two cores multiple times in quick succession. From experimentation, 5-10 invocations suffice in this preparation phase.

(2)/(3) Profiling for an anchor. Since the victim code execution is typically a subset of the

entire victim thread execution, we need to profile the execution of the victim thread to identify a consistent point of execution *just* before the target code to be faulted. We refer to this point of execution as a timing anchor, T_{anchor} to guide when to deliver the fault injection. Several software profiling techniques can be used to identify this timing anchor. In our case, we rely on instruction or data cache profiling techniques in recent work [87].

(4) Pre-fault delaying. Even with the timing anchor, in some attack scenarios, there may still be a need to finetune the exact delivery timing of the fault. In such cases, we can configure the attack thread to spin-loop with a predetermined number of loops before inducing the actual fault. The use of these loops consisting of *no-op* operations is essentially a technique to induce timing delays with high precision. For this stage of the attack, we term this delay before inducing the fault as F_{pdelay}.

(5)/(6) Delivering the fault. Given a base operating voltage F_{volt}, the attack thread will raise the frequency of the victim core (denoted as $F_{\text{freq\_hi}}$), keep that frequency for F_{dur} loops, and then restore the frequency to $F_{\text{freq\_lo}}$.

To summarize, for a successful CLKscrew attack, we can characterize the attacker's goal as the following sub-tasks. Given a victim code and a fault injection target point determined by T_{anchor}, the attacker has to find optimal values for the following parameters to maximize the odds of inducing the desired fault. We summarize the fault injection parameters required in Table 3.1.

$$F_{\theta|T_{\text{anchor}}} = \{F_{\text{volt}},\ F_{\text{pdelay}},\ F_{\text{freq\_hi}},\ F_{\text{dur}},\ F_{\text{freq\_lo}}\}$$

3.3.6 Isolation-Agnostic DVFS

To support execution of trusted code isolated from untrusted one, two leading industry technologies, ARM Trustzone [2] and Intel SGX [7], are widely deployed. They share a common characteristic in that they can execute both trusted and untrusted code on the *same* physical core, while relying on architectural features such as specialized instructions to support isolated execution. It is noteworthy that on such architectures, the voltage and frequency regulators typ-

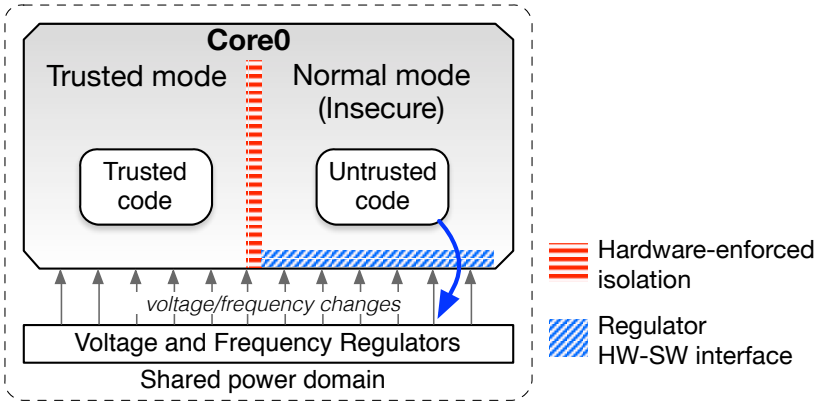

Figure 3.11: Regulators operate across security boundaries.

ically operate on domains that apply to cores as a whole (regardless of the security-sensitive processor execution modes), as depicted in Figure 3.11. With this design, any frequency or voltage change initiated by untrusted code inadvertently affects the trusted code execution, despite the hardware-enforced isolation. This, as we show in subsequent sections, poses a critical security risk.

> **Attack Enabler (General) #4**: Hardware regulators operate across security boundaries with no physical isolation.

3.4 TZ Attack #1: Inferring AES Keys

In this section, we show how AES [100] keys stored within Trustzone (TZ) can be inferred by lower-privileged code from outside Trustzone, based on the faulty ciphertexts derived from the erroneous AES encryption operations. Specifically, it shows how lower-privileged code can subvert the isolation guarantee by ARM Trustzone, by influencing the computation of higher-privileged code using the energy management mechanisms. The attack shows that the confidentiality of the AES keys that should have been kept secure in Trustzone can be broken.

Threat model. In our victim setup, we assume that there is a Trustzone app that provisions AES keys and stores these keys within Trustzone, inaccessible from the non-Trustzone (non-

secure) environment. The attacker can repeatedly invoke the Trustzone app from the non-secure environment to decrypt any given ciphertext, but is restricted from reading the AES keys directly from Trustzone memory due to hardware-enforced isolation. The attacker's goal is to infer the AES keys stored.

3.4.1 Trustzone AES Decryption App

For this case study, since we do not have access to a real-world AES app within Trustzone, we rely on a textbook implementation of AES as the victim app. We implement a AES decryption app that can be loaded within Trustzone. Without loss of generality, we restrict the decryption to 128-bit keys, operating on 16-bit plaintext and ciphertext. A single 128-bit encryption/decryption operation comprises 10 AES rounds, each of which is a composition of the four canonical sub-operations, named `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey` [100].

To load this app into Trustzone as our victim program, we use a publicly known Trustzone vulnerability[16] to overwrite an existing Trustzone syscall handler, `tzbsp_es_is_activated`, on our Nexus 6 device running an old firmware[8]. A non-secure app can then execute this syscall via an ARM Secure Monitor Call [42] instruction to invoke our decryption Trustzone app. This vulnerability serves the sole purpose of allowing us to load the victim app within Trustzone to simulate a AES decryption app in Trustzone. It plays no part in the attacker's task of interest – extracting the cryptographic keys stored within Trustzone. Having the victim app execute within Trustzone on a commodity device allows us to evaluate CLKscrew across Trustzone-enforced security boundaries in a practical and realistic manner.

3.4.2 Timing Profiling

As described in § 3.3.5, one of the crucial attack steps to ensure reliable delivery of the fault to a victim code execution is finding ideal values of F_{pdelay}. To guide this parameter discovery process, we need the timing profile of the Trustzone app performing a single AES encryption/decryption

[8]Firmware version is *shamu* `MMB29Q` (Feb, 2016)

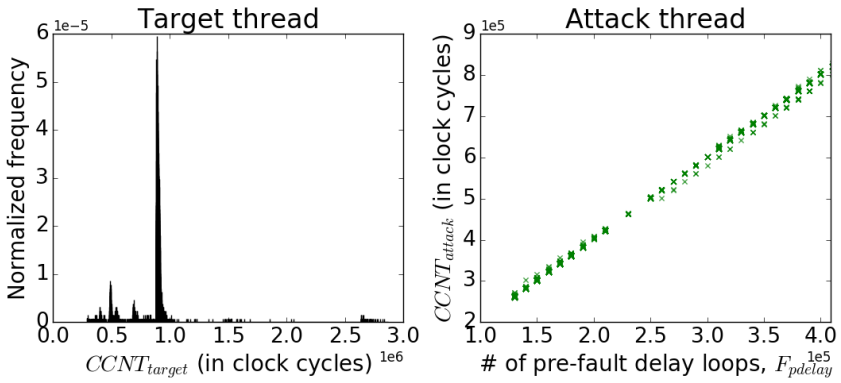

Figure 3.12: Execution duration (in clock cycles) of the victim and attack threads.

operation. ARM allows the use of hardware cycle counter (CCNT) to track the execution duration (in clock cycles) of Trustzone applications [3]. We enable this cycle counting feature within our custom kernel driver. With this feature, we can now measure how long it takes for our Trustzone app to decrypt a single ciphertext, even from the non-secure world.

Attack Enabler (TZ-specific) #5: Execution timing of code running in Trustzone can be profiled with hardware counters that are accessible outside Trustzone.

Using the hardware cycle counter, we track the duration of each AES decryption operation over about 13k invocations in total. Figure 3.12 (left) shows the distribution of the execution length of an AES operation. Each operation takes an average of 840k clock cycles with more than 80% of the invocations taking between 812k to 920k cycles. This shows that the victim thread does not exhibit too much variability in terms of its execution time.

Recall that we want to deliver a fault to specific region of the victim code execution and that the faulting parameter F_{pdelay} allows us to fine-tune this timing. Here, we evaluate the degree to which the use of *no-op* loops is useful in controlling the timing of the fault delivery. Using a fixed duration for the fault F_{dur}, we measure how long the attack thread takes in clock cycles for different values of the pre-fault delays F_{pdelay}. Figure 3.12 (right) illustrates a distinct linear relationship between F_{pdelay} and the length of the attack thread. This demonstrates that number of loops used in F_{pdelay} is a reasonably good proxy for controlling the execution timing of threads,

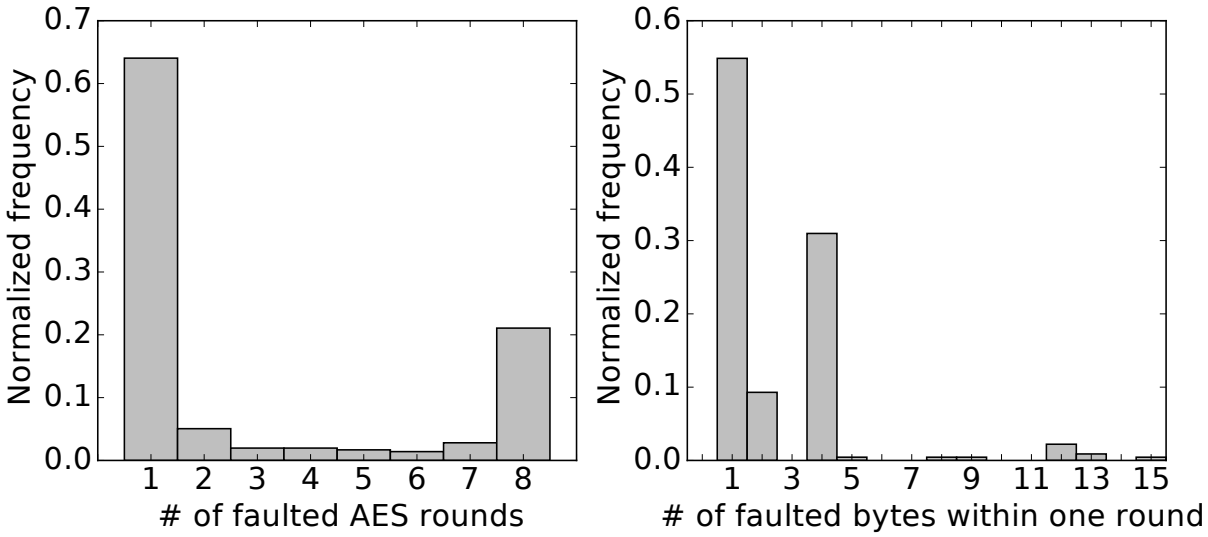

Figure 3.13: Fault model: Characteristics of observed faults induced by CLKscrew on AES operation.

and thus the timing of our fault delivery.

3.4.3 Fault Model

To detect if a fault is induced in the AES decryption, we add a check after the app invocation to verify that the decrypted plaintext is as expected. Moreover, to know exactly which AES round got corrupted, we add minimal code to track the intermediate states of the AES round and return this as a buffer back to the non-secure environment. A comparison of the intermediate states and their expected values will indicate the specific AES round that is faulted and the corrupted value. With these validation checks in place, we perform a grid search for the parameters for the faulting frequency, $F_{\text{freq\_hi}}$ and the duration of the fault, F_{dur} that can induce erroneous AES decryption results. From our empirical trials, we found that the parameters $F_{\text{freq\_hi}} = 3.69GHz$ and $F_{\text{dur}} = 680$ can most reliably induce faults to the AES operation.

For the rest of this attack, we assume the use of these two parameter values. By varying F_{pdelay}, we investigate the characteristics of the observed faults. A total of about 360 faults is observed. More than 60% of the faults are precise enough to affect exactly one AES round, as depicted in

Figure 3.13 (left). Furthermore, out of these faults that induce corruption in one AES round, more than half are sufficiently transient to cause random corruptions of exactly one byte, shown in Figure 3.13 (right). Being able to induce a one-byte random corruption to the intermediate state of an AES round is often used as a fault model in several physical fault injection works [154, 18].

3.4.4 Putting it together

Removing use of time anchor. Recall from § 3.3.5 that CLKscrew may require profiling for a time anchor to improve faulting precision. In this attack, we choose not to do so, because (1) the algorithm of the AES operation is fairly straightforward (one `KeyExpansion` round, followed by 10 AES rounds [100]) to estimate F_{pdelay}, and (2) the execution duration of the victim thread does not exhibit too much variability. The small degree of variability in the execution timing of both the attack and victim threads allows us to reasonably target specific AES rounds with a maximum error margin of one round.

Differential fault attack. Tunstall *et al.* present a differential fault attack (DFA) that infers AES keys based on pairs of correct and faulty ciphertext [154]. Since AES encryption is symmetric, we leverage their attack to infer AES keys based on pairs of correct and faulty plaintext. Assuming a fault can be injected during the seventh AES round to cause a single-byte random corruption to the intermediate state in that round, with a corrupted input to the eighth AES round, this DFA can reduce the number of AES-128 key hypotheses from the original 2^{128} to 2^{12}, in which case the key can be brute-forced in a trivial exhaustive search. We refer readers to Tunstall *et al.*'s work [154] for a full cryptanalysis for this fault model.

Degree of control of attack. To evaluate the degree of control we have over the specific round we seek to inject the fault in, we induce the faults using a range of F_{pdelay} and track which AES rounds the faults occur in. In Figure 3.14, each point represents a fault occurring in a specific AES round and when that fault occurs during the entire execution of the victim thread. We use the ratio of $CCNT_{\text{attack}}/CCNT_{\text{target}}$ as an approximation of latter. There are ten distinct clusters of faults corresponding to each AES round. Since $CCNT_{\text{target}}$ can be profiled beforehand and

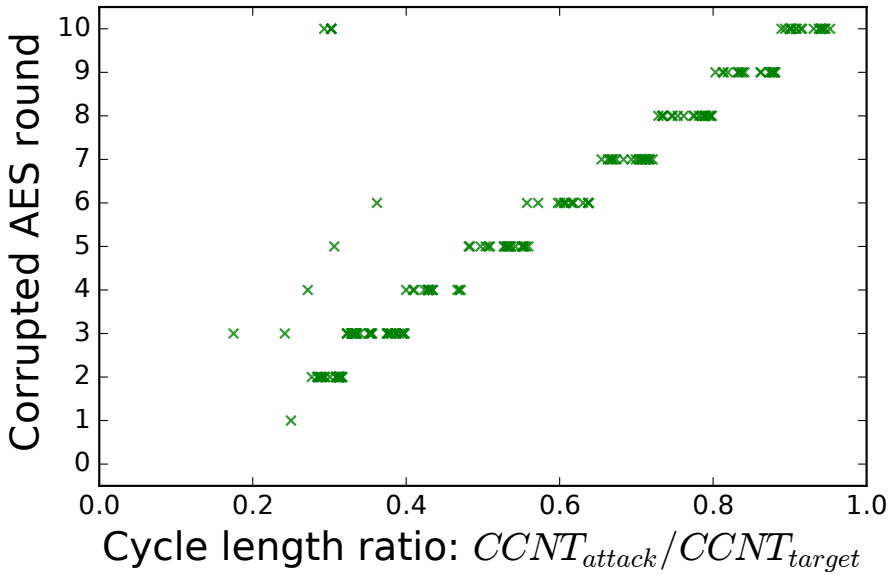

Figure 3.14: Controlling pre-fault delay, F_{pdelay}, allows us to control which AES round the fault affects.

$CCNT_{\text{attack}}$ is controllable via the use of F_{pdelay}, an attacker is able to control which AES round to deliver the fault to for this attack.

Actual attack. Given the faulting parameters, $F_{\theta,\text{ AES-128}} = \{F_{\text{volt}} = 1.055V,\ F_{\text{pdelay}} = 200k,\ F_{\text{freq\_hi}} = 3.69GHz,\ F_{\text{dur}} = 680,\ F_{\text{freq\_lo}} = 2.61GHz\}$, it took, on average, 20 faulting attempts to induce a one-byte fault to the input to the eighth AES round. Given the pair of this faulty plaintext and the expected one, it took Tunstall *et al.*'s DFA algorithm about 12 minutes on a $2.7GHz$ quad-core CPU to generate 3650 key hypotheses, one out of which is the AES key stored within Trustzone.

3.5 TZ Attack #2: Loading Self-Signed Apps

In this case study, we show how CLKscrew can subvert the RSA signature chain verification – the primary public-key cryptographic method used for authenticating the loading of firmware images into Trustzone. ARM-based SoC processors use the ARM Trustzone to provide a secure and isolated environment to execute security-critical applications like DRM *widevine* [53] trust-

let[9] and key management *keymaster* [43] trustlet. These vendor-specific firmware are subject to regular updates. These firmware update files consist of the updated code, a signature protecting the hash of the code, and a certificate chain. Before loading these signed code updates into Trustzone, the Trusted Execution Environment (TEE) authenticates the certificate chain and verifies the integrity of the code updates [120].

RSA Signature Validation. In the RSA cryptosystem[124], let N denote the modulus, d denote the private exponent and e denote the public exponent. In addition, we also denote the SHA-256 hash of code C as $H(C)$ for the rest of the section. To ensure the integrity and authenticity of a given code blob C, the code originator creates a signature Sig with its RSA private key: $Sig \leftarrow (H(C))^d \bmod N$. The code blob is then distributed together with the signature and a certificate containing the signing modulus N. Subsequently, the code blob C can be authenticated by verifying that the hash of the code blob matches the plaintext decrypted from the signature using the public modulus N: $Sig^e \bmod N == H(C)$. The public exponent is typically hard-coded to `0x10001`; only the modulus N is of interest here.

Threat model. The goal of the attacker is to provide an arbitrary attack app with a self-signed signature and have the TEE successfully authenticate and load this self-signed app within Trustzone. To load apps into Trustzone, the attackers can invoke the TEE to authenticate and load a given app into Trustzone using the `QSEOS_APP_START_COMMAND` [116] Secure Channel Manager[10] command. The attacker can repeatedly invoke this operation, but only from the non-secure environment.

3.5.1 Trustzone Signature Authentication

To formulate a CLKscrew attack strategy, we first examine how the verification of RSA signatures is implemented within the TEE. This verification mechanism is implemented within the

[9] Apps within Trustzone are sometimes referred to as *trustlets*.

[10] This is a vendor-specific interface that allows the non-secure world to communicate with the Trustzone secure world.

Algorithm 1 Given public key modulus N and exponent e, decrypt a RSA signature S. Return plaintext hash, H.

procedure DECRYPTSIG(S, e, N)
    $r \leftarrow 2^{2048}$
    $R \leftarrow r^2 \bmod N$
    $N_{rev} \leftarrow$ FLIPENDIANNESS(N)
    $r^{-1} \leftarrow$ MODINVERSE($r,\ N_{rev}$)
    $found\_first\_one\_bit \leftarrow false$
    for $i \in \{bitlen(e) - 1\ ..\ 0\}$ **do**
        if $found\_first\_one\_bit$ **then**
            $x \leftarrow$ MONTMULT($x,\ x,\ N_{rev},\ r^{-1}$)
            if $e[i] == 1$ **then**
                $x \leftarrow$ MONTMULT($x,\ a,\ N_{rev},\ r^{-1}$)
            end if
        else if $e[i] == 1$ **then**
            $S_{rev} \leftarrow$ FLIPENDIANNESS(S)
            $x \leftarrow$ MONTMULT($S_{rev},\ R,\ N_{rev},\ r^{-1}$)
            $a \leftarrow x$
            $found\_first\_one\_bit \leftarrow true$
        end if
    end for
    $x \leftarrow$ MONTMULT($x,\ 1,\ N_{rev},\ r^{-1}$)
    $H \leftarrow$ FLIPENDIANNESS(x)
    return H
end procedure

bootloader firmware. For the Nexus 6 in particular, we use the *shamu*-specific firmware image (`MOB31S`, dated Jan 2017 [46]), downloaded from the Google firmware update repository.

The RSA decryption function used in the signature verification is the function, DECRYPTSIG[11], summarized in Algorithm 1. At a high level, DECRYPTSIG takes, as input, a 2048-bit signature and the public key modulus, and returns the decrypted hash for verification. For efficient modular exponentiation, DECRYPTSIG uses the function MONTMULT to perform Montgomery multiplication operations [102, 81]. MONTMULT performs Montgomery multiplication of two inputs x and y with respect to the Montgomery *radix*, r [81] and modulus N as follows: MONTMULT($x,\ y,\ N,\ r^{-1}$) $\leftarrow x \cdot y \cdot r^{-1} \bmod N$.

In addition to the use of MONTMULT, DECRYPTSIG also invokes the function, FLIPENDIAN-

[11] DECRYPTSIG loads at memory address `0xFE8643C0`.

NESS[12], multiple times at lines 4, 14 and 21 of Algorithm 1 to reverse the contents of memory buffers. FLIPENDIANNESS is required in this implementation of DECRYPTSIG because the inputs to DECRYPTSIG are big-endian while MONTMULT operates on little-endian inputs. For reference, we outline the implementation of FLIPENDIANNESS in Algorithm 2.

Algorithm 2 Reverse the endianness of a memory buffer.

```
1:  procedure FLIPENDIANNESS(src)
2:      d ← 0
3:      dst ← {0}
4:      for i ∈ {0 .. len(src)/4 − 1} do
5:          for j ∈ {0 .. 2} do
6:              d ← (src[i ∗ 4 + j] | d) ≪ 8
7:          end for
8:          d ← src[i ∗ 4 + 3] | d
9:          k ← len(src) − i ∗ 4 − 4
10:         dst[k .. k + 3] ← d
11:     end for
12:     return dst
13: end procedure
```

3.5.2 Attack Strategy and Cryptanalysis

Attack overview. The overall goal of the attack is to deliver a fault during the execution of DECRYPTSIG such that the output of DECRYPTSIG results in the desired hash $H(C_A)$ of our attack code C_A. This operation can be described by Equation 3.2, where the attacker has to supply an attack signature $S_A{}'$, and fault the execution of DECRYPTSIG at runtime so that DECRYPTSIG outputs the intended hash $H(C_A)$. For comparison, we also describe the typical decryption operation of the *original* signature S to the hash of the original code blob, C in Equation 3.3.

$$\textbf{Attack}: \quad \textsc{DecryptSig}(S_A{}',\ e,\ N) \xrightarrow{fault} H(C_A) \tag{3.2}$$

$$\textbf{Original}: \quad \textsc{DecryptSig}(S,\ e,\ N) \longrightarrow H(C) \tag{3.3}$$

[12]FLIPENDIANNESS loads at memory address 0xFE868B20

For a successful attack, we need to address two questions: (a) At which portion of the runtime execution of DecryptSig($S_A{}',\ e,\ N$) do we inject the fault? (b) How do we craft $S_A{}'$ to be used as an input to DecryptSig?

3.5.2.1 Where to inject the runtime fault?

Target code of interest. The fault should target operations that manipulate the input modulus N, and ideally before the beginning of the modular exponentiation operation. A good candidate is the use of the function FlipEndianness at Line 4 of Algorithm 1. From experimentation, we find that FlipEndianness is especially susceptible to CLKscrew faults. We observe that N can be corrupted to a predictable N_A as follows:

$$N_{A,rev} \xleftarrow{fault} \text{FlipEndianness}(N)$$

Since $N_{A,rev}$ is N_A in reverse byte order, for brevity, we refer to $N_{A,rev}$ as N_A for the rest of the section.

Factorizable N_A. Besides being able to fault N to N_A, another requirement is that N_A must be factorizable. Recall that the security of the RSA cryptosystem depends on the computational infeasibility of factorizing the modulus N into its two prime factors, p and q [23]. This means that with the factors of N_A, we can derive the corresponding keypair $\{N_A,\ d_A,\ e\}$ using the Carmichael function in the procedure that is described in Razavi *et al.*'s work [123]. With this keypair $\{N_A,\ d_A,\ e\}$, the hash of the attack code C_A can then be signed to obtain the signature of the attack code, $S_A \leftarrow (H(C_A))^{d_A} \bmod N_A$.

We expect the faulted N_A to be likely factorizable due to two reasons: (a) N_A is likely a composite number of more than two prime factors, and (b) some of these factors are small. With sufficiently small factors of up to 60 bits, we use Pollard's ρ algorithm to factorize N_A and find them [97]. For bigger factors, we leverage the Lenstra's Elliptic Curve factorization Method (ECM) that has been observed to factor up to 270 bits [86]. Note that all we need for the attack

is to find a *single* N_A that is factorizable and reliably reproducible by the fault.

3.5.2.2 How to craft the attack signature $S_A{}'$?

Before we begin the cryptanalysis, we note that the attack signature $S_A{}'$ (an input to DecryptSig) is *not* the signed hash of the attack code, S_A (private-key encryption of the $H(C_A)$). We use $S_A{}'$ instead of S_A primarily due to the pecularities of our implementation. Specifically, this is because the operations that follow the injection of the fault also use the parameter values derived before the point of injected fault. Next, we sketch the cryptanalysis of delivering a fault to DecryptSig to show how the desired $S_A{}'$ is derived, and demonstrate why $S_A{}'$ is *not trivially* derived the same way as S_A.

Cryptanalysis. The goal is to derive $S_A{}'$ (as input to DecryptSig) given an expected corrupted modulus N_A, the original vendor's modulus N, and the signature of the attack code, S_A. For brevity, all line references in this section refer to Algorithm 1. The key observation is that after being derived from FlipEndianness at Line 4, N_{rev} is next used by MontMult at Line 15. Line 15 marks the beginning of the modular exponentiation of the input signature, and thus, we focus our analysis here.

First, since we want DecryptSig$(S_A{}',\ e,\ N)$ to result in $H(C_A)$ as dictated by Equation 3.2, we begin by analyzing the invocation of DecryptSig that will lead to $H(C_A)$. If we were to run DecryptSig with inputs S_A and N_A, DecryptSig$(S_A,\ e,\ N_A)$ should output $H(C_A)$. Based on the analysis of this invocation of DecryptSig, we can then characterize the output, $x_{desired}$, of the operation at Line 15 of DecryptSig$(S_A,\ e,\ N_A)$ with Equation 3.4. We note that the modular inverse of r is computed based on N_A at Line 5, and so we denote this as r_A^{-1}.

$$x_{desired} \leftarrow S_A \cdot (r^2 \bmod N_A) \cdot r_A^{-1} \bmod N_A \tag{3.4}$$

Next, suppose our CLKscrew fault is delivered in the operation DecryptSig$(S_A{}',\ e,\ N)$ such that N is corrupted to N_A at Line 4. We note that while N is faulted to N_A at Line 4, subsequent

instructions continue to indirectly use the original modulus N because R is derived based on the uncorrupted modulus N at Line 3. Herein lies the complication. The attack signature $S_A{}'$ passed into DecryptSig gets converted to the Montgomery representation at Line 15, where *both* moduli are used:

$$x_{fault} \leftarrow \text{MontMult}(S_A{}',\ r^2 \bmod N,\ N_A,\ r_A^{-1})$$

We can then characterize the output, x_{fault}, of the operation at the same Line 15 of a faulted DecryptSig$(S_A{}',\ e,\ N)$ as follows:

$$x_{fault} \leftarrow S_A{}' \cdot (r^2 \bmod N) \cdot r_A^{-1} \bmod N_A \tag{3.5}$$

By equating $x_{fault} = x_{desired}$ (*i.e.* equating results from (3.4) and (3.5)), we can reduce the problem to finding $S_A{}'$ for constants $K = (r^2 \bmod N) \cdot r_A^{-1}$ and $x_{desired}$, such that:

$$S_A{}' \cdot K \bmod N_A \equiv x_{desired} \bmod N_A$$

Finally, subject to the condition that $x_{desired}$ is divisible[13] by the greatest common divisor of K and N_A, denoted as $gcd(K,\ N_A)$, we use the Extended Euclidean Algorithm[14] to solve for the attack signature $S_A{}'$, since there exists a constant y such that $S_A{}' \cdot K + y \cdot N_A = x_{desired}$. In summary, we show that the attack signature $S_A{}'$ (to be used as an input to DecryptSig$(S_A{}',\ e,\ N)$) can be derived from N, N_A and S_A.

[13]We empirically observe that $gcd(K,\ N_A) = 1$ in our experiments, thus making $x_{desired}$ trivially divisible by $gcd(K,\ N_A)$ for our purpose.

[14]The Extended Euclidean Algorithm is commonly used to compute, besides the greatest common divisor of two integers a and b, the integers x and y where $ax + by = gcd(a,\ b)$.

3.5.3 Timing Profiling

Each trustlet app file on the Nexus 6 device comes with a certificate chain of two RSA certificates (and signatures). Before loading an app into Trustzone, the loader will validate the signatures of the two certificates and the metadata that comes with the app [120]. The DecryptSig code that decrypts the RSA signature gets invoked four times in total. By incrementally corrupting each certificate and then invoking the loading of the app with the corrupted chain, we measure the operation of validating one signature to take about 270 million cycles on average. We extract the target function FlipEndianness from the binary firmware image and execute it in the non-secure environment to measure its length of execution. We profile its invocation on a 256-byte buffer (the size of the 2048-bit RSA modulus) to take on average 65k cycles.

To show the feasibility of our attack, we choose to attack the fourth invovation of the DecryptSig code during the whole app loading process. This requires a *very precise* fault to be induced within in a 65k-cycle-long targeted period within an entire chain validation operation that takes 270 million x 4 = 1.08 billion cycles, a duration that is four orders of magnitude longer than the targeted period. Due to the degree of precision needed, it is thus crucial to find a way to determine a reliable time anchor (see Steps ❷ / ❸ in § 3.3.5) to guide the delivery of the fault.

Cache profiling To determine approximately which region of code is being executed during the chain validation at any point in time, we leverage side-channel-based cache profiling attacks that operate across cores. Since we are profiling code execution within Trustzone in a separate core, we use recent advances in the cross-core instruction- and data-based *Prime+Probe*[15] cache attack techniques [87, 57, 177]. We observe that the cross-core profiling of the instruction-cache usage of the victim thread is more reliable than that of the data-cache counterpart. As such, we adapt the instruction-based *Prime+Probe* cache attack for our profiling stage.

Within the victim code, we first identify the code address we want to monitor, and then compute the set of memory addresses that is congruent to the cache set of our monitored code address.

[15]Another prevalent class of cross-core cache attacks is the *Flush+Reload* [175] cache attacks. We cannot use the *Flush+Reload* technique to profile Trustzone execution because *Flush+Reload* requires being able to map addresses that are shared between Trustzone and the non-secure environment. Trustzone, by design, prohibits that.

Since we are doing instruction-based cache profiling, we need to rely on executing instructions instead of memory read operations. We implement a loop within the fault injection thread to continuously execute dynamically generated dummy instructions in the cache-set-congruent memory addresses (the $Prime$ step) and then timing the execution of these instructions (the $Probe$ step) using the clock cycle counter. We determine a threshold for the cycle count to indicate that the associated cache lines have been evicted. The eviction patterns of the monitored cache set provides an indication that the monitored code address has been executed.

> **Attack Enabler (TZ-specific) #6**: Memory accesses from the non-secure world can evict cache lines used by Trustzone code, thereby enabling *Prime+Probe*-style execution profiling of Trustzone code.

While we opt to use the *Prime+Probe* cache profiling strategy in our attack, there are alternate side-channel-based profiling techniques that can also be used to achieve the same effect. Other microarchitectural side channels like branch predictors, pipeline contention, prefetchers, and even voltage and frequency side channels can also conceivably be leveraged to profile the victim execution state. Thus, more broadly speaking, the attack enabler #6 is the presence of microarchitectural side channels that allows us to profile code for firing faults.

App-specific timing feature. For our timing anchor, we want a technique that is more fine-grained. We devise a novel technique that uses the features derived from the eviction timing to create a proxy for profiling program phase behavior. First, we maintain a global incrementing count variable as an approximate time counter in the loop. Then, using this counter, we track the duration between consecutive cache set evictions detected by our *Prime+Probe* profiling. By treating this series of *eviction gap duration* values, g, as a time-series stream, we can approximate the execution profile of the chain validation code running within Trustzone.

We plot a snapshot of the cache profile characterizing the validation of the fourth and final certificate in Figure 3.15. We observe that the beginning of each certification validation is preceded by a large spike of up to 75,000 in the g values followed by a secondary smaller spike. From experimentation, we found that FlipEndianness runs after the second spike. Based on this ob-

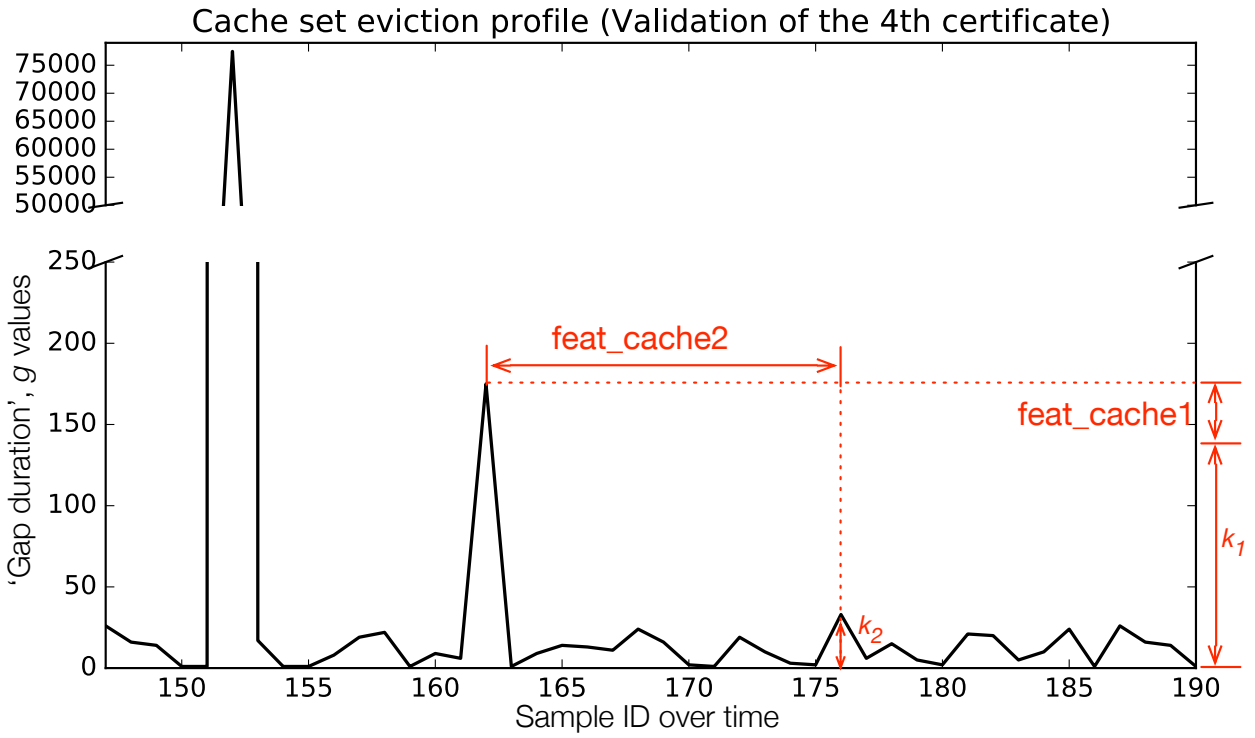

Figure 3.15: Cache eviction profile snapshot with cache-based features.

servation, we change the profiling stage of the attack thread to track two hand-crafted timing features to characterize the instantaneous state of victim thread execution.

Timing anchor. We annotate the two timing features on the cache profile plot in Figure 3.15. The first feature, *feat_cache1*, tracks the length of the second spike minus a constant k_1. The second feature, *feat_cache2*, tracks the cumulative total of g after the second spike, until the $g > k_2$. We use a value of $k_1 = 140$ and $k_2 = 15$ for our experiments. By continuously monitoring values of g after the second spike, the timing anchor is configured to be the point when $g > k_2$.

To evaluate the use of this timing anchor, we need a means to assess when and how the specific invocation of the FlipEndianness is faulted. First, we observe that the memory buffer used to store N_{rev} is hard-coded to an address `0x0FC8952C` within Trustzone, and this buffer is not zeroed out after the validation of each certificate. We downgrade the firmware version to `MMB29Q` (Feb, 2016), so that we can leverage a Trustzone memory safety violation vulnerability [16] to

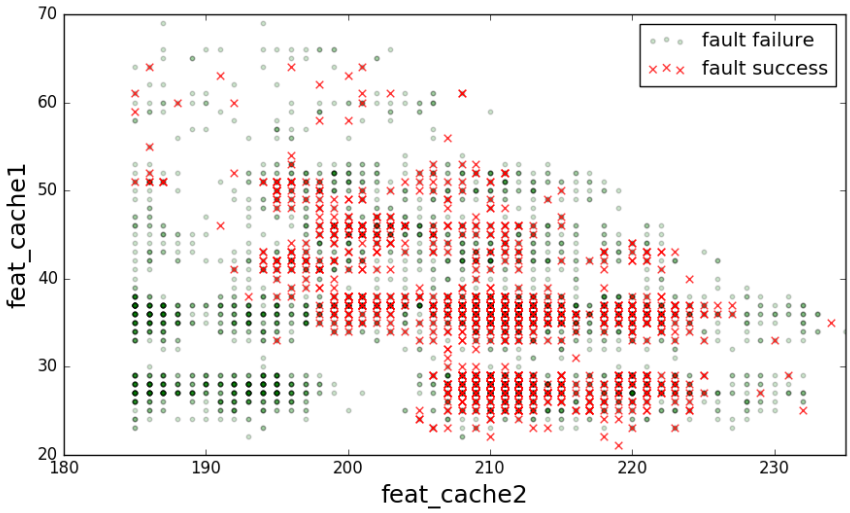

Figure 3.16: Observed faults using the timing features.

access the contents of N_{rev} after the fourth invocation of DecryptSig code[16]. Note that this does not affect the normal operation of the chain validation because the relevant code sections for these operations is identical across version `MMB29Q` (Feb, 2016) and `MOB31S` (Jan, 2017).

With this timing anchor, we perform a grid search for the faulting parameters, $F_{\text{freq\_hi}}$, F_{dur} and F_{pdelay} that can best induce faults in FlipEndianness. The parameters $F_{\text{freq\_hi}} = 3.99GHz$ and $F_{\text{dur}} = 1$ are observed to be able to induce faults in FlipEndianness reliably. The value of the pre-fault delay parameter F_{pdelay} is crucial in controlling the type of byte(s) corruption in the target memory buffer N_{rev}. With different values of F_{pdelay}, we plot the observed faults and failed attempts based on the values of *feat_cache1* and *feat_cache2* in Figure 3.16. Each faulting attempt is considered a success if any bytes within N_{rev} are corrupted during the fault.

Adaptive pre-delay. While we see faults within the target buffer, there is some variability in the position of the fault induced within the buffer. In Figure 3.17, each value of F_{pdelay} is observed to induce faults across all parts of the buffer. To increase the precision in faulting, we modify the fault to be delivered based on an adaptive F_{pdelay}.

[16]We are solely using this vulnerability to speed up the search for the faulting parameters. They can be replaced by more accurate and precise side-channel-based profiling techniques.

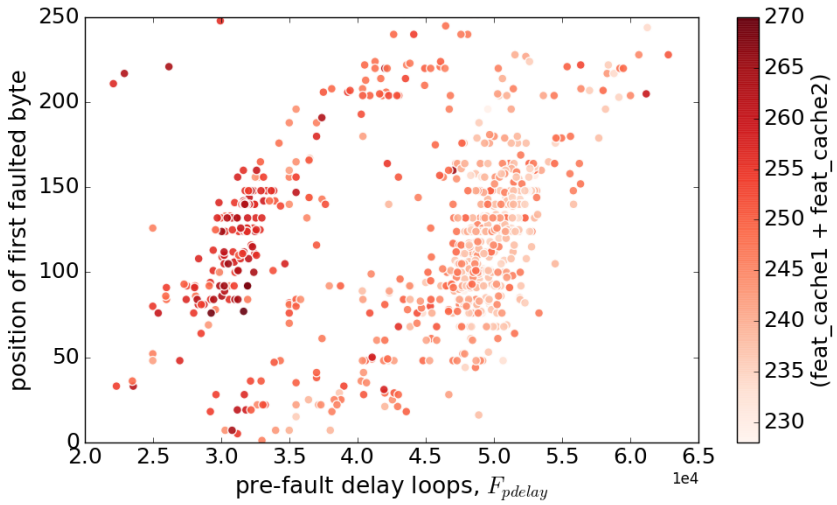

Figure 3.17: Variability of faulted byte(s) position.

3.5.4 Fault Model

Based on the independent variables *feat_cache1* and *feat_cache2*, we build linear regression models to predict F_{pdelay} that can best target a fault at an intended position within the N_{rev} buffer. During each faulting attempt, F_{pdelay} is computed only when the timing anchor is detected. To evaluate the efficacy of the regression models, we collect all observed faults with the goal of injecting a fault at byte position 141. Figure 3.18 shows a significant clustering of faults around positions 140 - 148.

More than 80% of the faults result in 1-3 bytes being corrupted within the N_{rev} buffer. Many of the faulted values suggest that instructions are skipped when the fault occurs. An example of a fault within a segment of the buffer is having corrupted the original byte sequence from `0xa777511b` to `0xa7777777`.

3.5.5 Putting it together

We use the following faulting parameters to target faults to specific positions within the buffer: $F_{\theta,\,\text{RSA}} = \{F_{\text{volt}} = 1.055V,\ F_{\text{pdelay}} = \text{adaptive},\ F_{\text{freq\_hi}} = 3.99GHz,\ F_{\text{dur}} = 1,\ F_{\text{freq\_lo}} = 2.61GHz\}$.

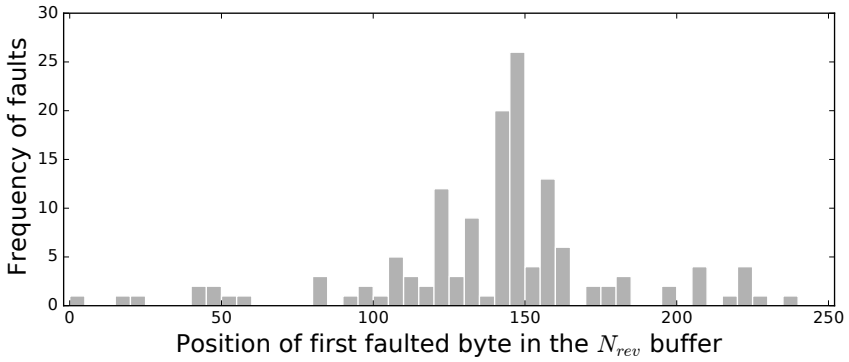

Figure 3.18: Histogram of observed faults and where the faults occur. The intended faulted position is 141.

Factorizable modulus N_A. About 20% of faulting attempts (1153 out of 6000) result in a successful fault within the target N_{rev} buffer. This set of faulted N values consists of 805 unique values, of which 38 (4.72%) are factorizable based on the algorithm described in § 3.5.2. For our attack, we select one of the factorizable N_A, where two bytes at positions 141 and 142 are corrupted. We show an example of this faulted and factorizable modulus in Appendix A.1.

Actual attack. Using the above selected N_A, we embed our attack signature $S_A{}'$ into the *widevine* trustlet. Then we conduct our CLKscrew faulting attempts while invoking the self-signed app. On average, we observe one instance of the desired fault in 65 attempts.

3.6 Discussion and Related Works

3.6.1 Applicability to other Platforms

Several highlighted attack enablers in preceding sections apply to other leading architectures. In particular, the entire industry is increasingly moving or has moved to fine-grained energy management designs that separate voltage/frequency domains for the cores. We leave the exploration of these architectures to future research.

Intel. Intel's recent processors are designed with the base clock separated from the other clock domains for more scope of energy consumption optimization [59, 72]. This opens up pos-

sibilities of overclocking on Intel processors [178]. Given these trends in energy management design on Intel hardware and the growing prevalence of Intel's Secure Enclave SGX [69], a closer look at whether the security guarantees still hold is warranted.

ARMv8. The ARMv8 devices adopt the ARM *big.LITTLE* design that uses non-symmetric cores (such as the "big" Cortex-A15 cores, and the "LITTLE" Cortex-A7 cores) in same system [73]. Since these cores are of different architectures, they exhibit different energy consumption characteristics. It is thus essential that they have separate voltage/frequency domains. The use of separate domains, like in the 32-bit ARMv7 architecture explored in this work, expose the 64-bit ARMv8 devices to similar potential dangers from the software-exposed energy management mechanisms.

Cloud computing providers. The need to improve energy consumption does not just apply to user devices; this extends even to cloud computing providers. Since 2015, Amazon AWS offers EC2 VM instances [15] where power management controls are exposed within the virtualized environment. In particular, EC2 users can fine-tune the processor's performance using P-state and C-state controls [6]. This warrants further research to assess the security ramifications of such user-exposed energy management controls in the cloud environment.

3.6.2 Hardware-Level Defenses

Operating limits in hardware. CLKscrew requires the hardware regulators to be able to push voltage/frequency past the operating limits. To address this, hard limits can be enforced within the regulators in the form of additional limit-checking logic or e-fuses [125]. However, this can be complicated by three reasons. First, adding such enforcement logic in the regulators requires making these design decisions very early in the hardware design process. However, the operational limits can only be typically derived through rigorous electrical testing in the post-manufacturing process. Second, manufacturing process variations can change operational limits even for chips of the same designs fabricated on the same wafer. Third, these hardware regulators are designed to work across a wide range of SoC processors. Imposing a one-size-fits-all range of

limits is challenging because SoC-specific limits hinder the portability of these regulators across multiple SoC. For example, the PMIC found on the Nexus 6 is also deployed on the Galaxy Note 4.

Separate cross-boundary regulators. Another mitigation is to maintain different power domains across security boundaries. This entails using a separate regulator when the isolated environment is active. This has two issues. First, while trusted execution technologies like Trustzone and SGX separate execution modes for security, the different modes continue to operate on the same core. Maintaining separate regulators physically when the execution mode switches can be expensive. Second, DVFS components typically span across the system stack. If the trusted execution uses dedicated regulators, this implies that a similar cross-stack power management solution needs to be implemented within the trusted mode to optimize energy consumption. Such an implementation can impact the runtime of the trusted mode and increase the complexity of the trusted code.

Redundancy/checks/randomization. To mitigate the effects of erroneous computations due to induced faults, researchers propose redesigning the application core chip with additional logic and timing redundancy [12], as well as recovery mechanisms [65]. Also, Bar-El *et al.* suggest building duplicate microarchitectural units and encrypting memory bus operations for attacks that target memory operations [12]. Luo *et al.* present a clock glitch detection technique that monitors the system clock signal using another higher frequency clock signal [90]. While many of these works are demonstrated on FPGAs [157] and ASICs [139], it is unclear how feasible it is on commodity devices and how much chip area and runtime overhead it adds. Besides adding redundancy, recent work proposes adding randomization using reconfigurable hardware as a mitigation strategy [158].

3.6.3 Software-Level Defenses

Randomization. Since CLKscrew requires some degree of timing precision in delivering the faults, one mitigation strategy is to introduce randomization (via no-op loops) to the runtime

execution of the code to be protected. However, we note that while this mitigates against attacks without a timing anchor (AES attack in § 3.4), it may have limited protection against attacks that use forms of runtime profiling for the timing guidance (RSA attack in § 3.5).

Redundancy and checks in security-sensitive code. Several software-only defenses propose compiling code with checksum integrity verification and execution redundancy (executing sensitive code multiple times) [12, 13]. Security-sensitive code, such as signature verification code, executing within TEEs can be rewritten and recompiled to include additional checks and redundancy in computation. While these defenses may be deployed on systems requiring high dependability, they are not typically deployed on commodity devices like phones because they can impact energy efficiency.

TEE-mediated software interfaces. One pertinent design issue highlighted in this work is the fact that the hardware regulators can be accessed without restrictions from privileged software running outside the TEEs. A possible solution is to restrict all non-TEE software access to hardware regulators. This means that all frequency/voltage changes initiated from DVFS software running outside TEEs have to incur a context switch to the TEE to effect the changes. Depending on how frequent DVFS regulates the runtime frequency and voltage, this may incur non-trivial performance overheads.

3.6.4 Subverting Cryptography with Faults

Boneh *et al.* offer the first DFA theoretical model to breaking various cryptographic schemes using injected hardware faults [24]. Subsequently, many researchers demonstrate physical fault attacks using a range of sophisticated fault injection equipment like laser [27, 39] and heat [54]. Compared to these attacks including all known undervolting [14, 105] and overclocking [22] ones, CLKscrew does not need physical access to the target devices, since it is initiated entirely from software. CLKscrew is also the first to demonstrate such attacks on a commodity device. We emphasize that while CLKscrew shows how faults can break cryptographic schemes, it does so to highlight the dangers of hardware regulators exposing software-access interfaces, especially

across security trust boundaries.

3.6.5 Relation to Rowhammer Fault Attacks

Kim *et al.* first present reliability issues with DRAM memory [79] (dubbed the "Rowhammer" problem). Since then, many works use the Rowhammer issue to demonstrate the dangers of such software-induced hardware-based transient bit-flips in practical scenarios ranging from browsers [56], virtualized environments [123], privilege escalation on Linux kernel [126] and from Android apps [156]. Like Rowhammer, CLKscrew is equally pervasive. However, CLKscrew is the manifestation of a different attack vector relying on software-exposed energy management mechanisms. The complexity of these cross-stack mechanisms makes any potential mitigation against CLKscrew more complicated and challenging. Furthermore, unlike Rowhammer that corrupts DRAM *memory*, CLKscrew targets *microarchitectural* operations. While we use CLKscrew to induce faults in memory contents, CLKscrew can conceivably affect a wider range of computation in microarchitectural units other than memory (such as caches, branch prediction units, arithmetic logic units and floating point units).

3.6.6 Relation to Meltdown/Spectre Side-Channel Attacks

Like CLKscrew , recently published Meltdown [88] and Spectre [83] attacks are glaring examples of security flaws introduced by system architects seeking to optimize performance. Using cache timing side channels, the Meltdown and Spectre attacks demonstrate how sensitive memory can be exposed by probing residual cache states that result from aggressive speculative execution. They combine the use of side-channel attacks with novel manipulation of aggressive speculative execution states of both Intel and ARM processors. At a high level, all these three hardware-oriented attacks bear testament to the importance of approaching architecture design with a full-system approach. Myopic and unduly-performance-focused architecture design can be dangerous – security risks can occur when different system components interact in unanticipated ways. As microarchitectural attacks, like Meltdown/Spectre, CLKscrew is equally pervasive.

In terms of degree of security impact, CLKscrew extends beyond Meltdown/Spectre. Besides allowing attackers to perform unauthorized memory reads (Meltdown/Spectre break only confidentiality), CLKscrew also breaks code integrity where it allows attackers to load and execute unauthorized code.

3.7 Conclusions

As researchers and practitioners embark upon increasingly aggressive cooperative hardware-software mechanisms with the aim of improving energy efficiency, this work shows, for the first time, that doing so may create serious security vulnerabilities. With only publicly available information, we have shown that the sophisticated energy management mechanisms used in state-of-the-art mobile SoCs are vulnerable to confidentiality, integrity and availability attacks. Our CLKscrew attack is able to subvert even hardware-enforced security isolation and does not require physical access, further increasing the risk and danger of this attack vector.

While we offer proof of attackability in this paper, the attack can be improved, extended and combined with other attacks in a number of ways. For instance, using faults to induce specific values at exact times (as opposed to random values at approximate times) can substantially increase the power of this technique. Furthermore, CLKscrew is the tip of the iceberg: more security vulnerabilities are likely to surface in emerging energy optimization techniques, such as finer-grained controls, distributed control of voltage and frequency islands, and near/sub-threshold optimizations.

Our analysis suggests that there is unlikely to be a single, simple fix, or even a piecemeal fix, that can entirely prevent CLKscrew style attacks. Many of the design decisions that contribute to the success of the attack are supported by practical engineering concerns. In other words, the root cause is not a specific hardware or software bug but rather a series of well-thought-out, nevertheless security-oblivious, design decisions. To prevent these problems, a coordinated full system response is likely needed, along with accepting the fact that some modest cost increases

may be necessary to harden energy management systems. This demands research in a number of areas such as better Computer Aided Design (CAD) tools for analyzing timing violations, better validation and verification methodology in the presence of DVFS, architectural approaches for DVFS isolation, and authenticated mechanisms for accessing voltage and frequency regulators. As system designers work to invent and implement these protections, security researchers can complement these efforts by creating newer and exciting attacks on these protections.

4

HEISENBYTE: Stemming Code Reuse Exploits with Destructive Code Reads

Assistive virtualization hardware features enable timely and transparent mediation of read operations into executable memory.

The destructive code read primitive protects closed-source software and just-in-time compiled code against memory disclosure exploits.

Vulnerabilities that disclose executable memory pages enable a new class of powerful code reuse attacks that build the attack payload at runtime. These make "moving target" defenses like fine-grained randomization inadequate for avoiding exploitation. In this chapter, we present *Heisenbyte*, a system to protect against such memory disclosure attacks. Central to Heisenbyte is the concept of *destructive code reads* – code is garbled right after it is read. Garbling the code after reading it takes away from the attacker her ability to leverage memory disclosure bugs in both static code and dynamically generated just-in-time code. By leveraging existing virtualization support, Heisenbyte's novel use of destructive code reads sidesteps the problem of incomplete binary disassembly in binaries, and extends protection to close-sourced COTS binaries, which are two major limitations of prior solutions against memory disclosure vulnerabilities. Our experiments demonstrate that Heisenbyte can tolerate some degree of imperfect static analysis in disassembled binaries, while effectively thwarting dynamic code reuse exploits in both static and JIT code, at a modest 1.8% average runtime overhead due to virtualization and 16.5% average overhead due to the destructive code reads.

4.1 Introduction

In the last decade, with the widespread use of data execution protection, attackers have turned to reusing code snippets from existing binaries to craft attacks. To perform these code reuse attacks, the attacker has to "see" the code so that she can find the "gadgets" necessary to craft the attack payload. One effective solution, until very recently, has been fine-grained randomization. The idea is to shuffle the code to blind the attacker from seeing the code layout in memory. The assumption behind this approach is that without knowledge of the code layout, the attacker cannot craft payloads. However, as demonstrated by Snow *et al.* in 2013, it is feasible and practical to scan for ROP gadgets at runtime and construct a *dynamic* JIT attack payload [133]. The attack by Snow *et al.* undermines the use of fine-grained randomization as an mitigation against ROP attacks.

To counter this new threat, researchers have revived the idea of execute-only memory (XoM) [150]. This approach involves preventing programs from reading executable memory using general purpose memory access instructions. One challenge in realizing these systems is that legacy binaries and compilers often intersperse code and data (*e.g.* jump tables) in executable memory pages. Thus, the wholesale blinding of executable memory at page granularity is not an option. To tackle this issue, researchers have used static compilation techniques to separate code and data [33]. However, this solution does not work well in the absence of source code, for instance, when utilizing legacy binaries. In fact, separating data from code has been shown to be provably undecidable [162]. Another complication in realizing the XoM concept arises from web browsers' use of JIT code where data becomes dynamically generated code. This has been shown to be a significant attack surface for browsers [9, 136].

In this work, we propose a new concept to deal with memory disclosure attacks. Unlike XoM and XoM-inspired systems, which aim to completely prevent reads to executable memory, a task beset with many practical difficulties, we allow executable memory to be read, but make them unusable as code after being read. In essence, in our model, as soon as the code is read using a general-purpose memory dereferencing instruction, the copy of code in memory is gar-

bled. Manipulating executable memory in this manner allows legitimate code to execute without false-positives and false-negatives, while servicing legitimate memory read operations for data embedded in the code. We term our special code read operations as *destructive code reads*.

We implement our new code read mechanism by leveraging existing virtualization hardware support on commodity processors. We term our system *Heisenbyte*[1].

Our experiments demonstrate that Heisenbyte can thwart the use of memory disclosure attacks on executable memory, both from static program binaries and dynamically generated JIT code on a production Windows 7 machine at a modest average runtime overhead of 16.5% and 18.3% on virtualized and non-virtualized systems respectively.

This chapter makes the following contributions:

1. We conceptualize a novel destructive code read primitive that tolerates legitimate data reads in executable memory while preventing the same data from being used as code in a dynamic code reuse attack.
2. We implement Heisenbyte to realize this destructive code read operation in practice on contemporary commodity systems.
3. We demonstrate its utility in preventing attacks that use memory disclosure bugs on both static program binaries and dynamic JIT code in close-sourced COTS binaries.

4.2 Background

In this section, we describe the steps of a typical *dynamic* code reuse attack. Since the use of memory disclosure vulnerabilities is crucial in a dynamic code reuse attack (*cf.* static code reuse attacks [130, 21]), we will focus on techniques that aim to thwart executable memory disclosures. We also cover the assumptions of the threat model and the capabilities of the adversary.

[1]A tribute to renowned physicist, Werner Heisenberg, who observed that the act of observing a system inevitably changes its state in quantum mechanics.

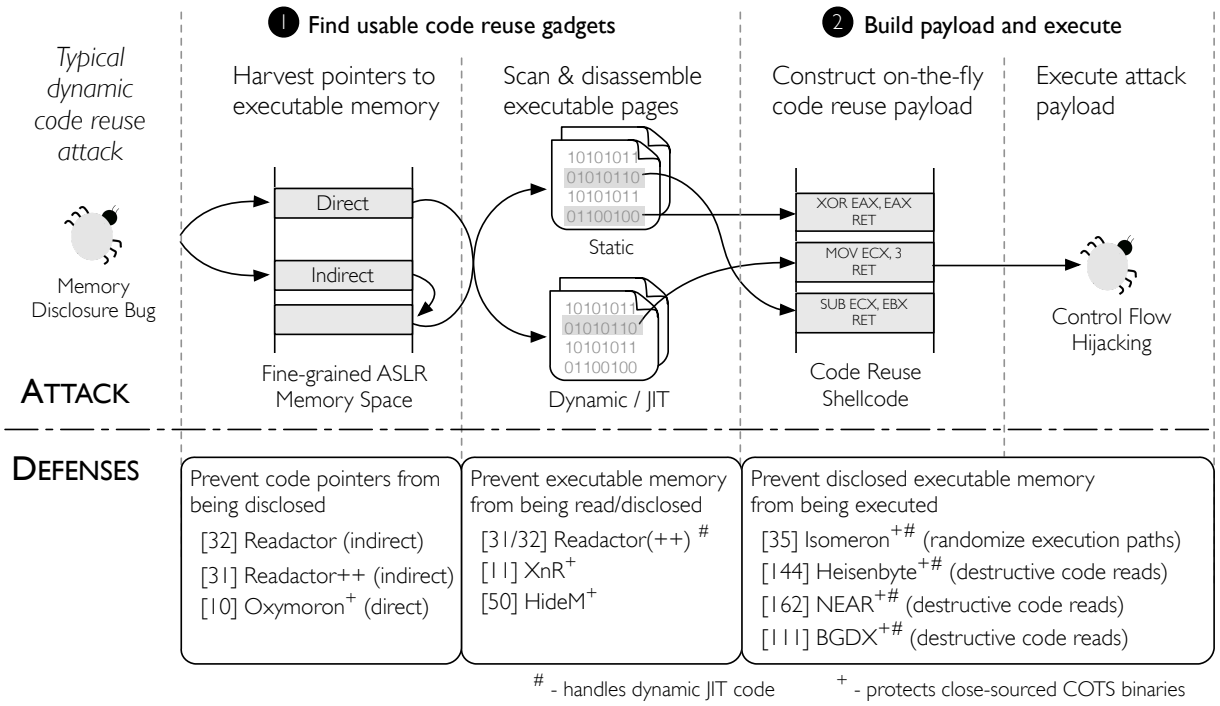

Figure 4.1: TOP: Stages of a code reuse attack that constructs its payload on-the-fly using executable memory found with a memory disclosure bug. BOTTOM: Taxonomy of defenses grouped by their defense strategy.

4.2.1 Dynamic Code Reuse Attacks

In the top half of Figure 4.1, we show the stages of a typical code reuse attack, and the sub-steps within each stage. Typical dynamic code reuse attacks comprise two stages, namely ❶ the search for usable code reuse gadgets in either static code [133] or dynamic JIT code [9], and ❷ building the payload on-the-fly and then redirecting execution to the payload.

To gather code reuse gadgets for the dynamic exploit, an adversary needs to first uncover memory pages that are executable. Note that a trivial linear scan of the memory cannot be used as it is likely to trigger a page fault or access unmapped guard pages placed randomly in the address space. Therefore, to craft a stable exploit, the adversary has to first gather pointers to the memory pages marked as executable. These pointers can be direct branches into executable memory or indirect pointeres residing in data pages but pointing to code memory.

With the list of the pointers to executable memory, the adversary can then invoke a memory disclosure bug repeatedly (without crashing the vulnerable program) to scan and disassemble the memory pages looking for suitable code reuse gadgets. The next step involves stringing the locations of the gadgets together in an exploit payload, and finally redirecting execution to this payload using another control flow hijacking vulnerability.

4.2.2 Previous Works

The first category of defenses focuses on protecting the code pointers and preventing them from being disclosed, stifling the attack as earlier as possible. Oxymoron hides the direct code pointers by generating randomized code that does not have direct references to code pages [10]. However, besides using direct references to code pages, adversaries can use indirect code references that reside in stack and heap. Readactor and Readactor++ address this by masking the indirect code references with executable trampolines that are protected by hardware virtualization feature [33, 32].

The next set of works introduces the concept of execute-only memory implemented in software. This is designed to prevent executable memory from being disclosed directly through memory read operations, consequently removing the adversary's ability to scan and locate suitable code reuse sequences for the attack. To achieve this, these works have to separate legitimate data from executable sections of programs, and distinguish at runtime between code execution and data read operations in executable memory.

XnR configures executable pages to be non-executable and augments the page fault handler to mediate illegal reads into code pages [11], but it is susceptible to disclosure attacks via indirect code references. HideM leverages the spilt-TLB architecture on AMD processors to transparently prevent code from being read by memory dereferencing operations [51]. The use of split-TLB limits its ability to remove all data from the executable sections, and inevitably exposes these data remnants to being used in attacks. Readactor relies on compiler-based techniques to separate legitimate data from code in programs and uses hardware virtualization support to enforce

execute-only memory [33].

Unlike the previous defenses that *protect* the executable memory from illegal memory reads, the third group of works *tolerates* the disclosure of executable memory contents in attacks. It shifts the focus of the defense strategy to preventing any discovered gadgets from the earlier attack stages from being used in later stages of attacks.

This paradigm shift in protection strategy addresses a key limitation of past execute-only memory defenses – legacy binary compatibility, since such defenses require clean and accurate separation of code and data. While recent work by Andriesse *et al.* shows interspersed code and data are non-existent in programs compiled with *gcc* or *clang* compilers [8], the authors also highlighted that many *MSVC*-compiled programs on Windows have intermingled code and data. It is noteworthy that since many library code is highly optimized and contains hand-crafted assembly, having both code and data in the execution memory sections is no surprise, and is thus still a huge problem in applying execute-only memory-based defenses.

Belonging to this class of defenses, Isomeron probabilistically impedes the use of the discovered gadgets by randomizing the control flow at runtime specifically for dynamically generated code [36]. The work described in this chapter, Heisenbyte [147], also falls into this third category of defenses. While most works either enforce execute-only code memory or hide important static code contents from adversaries, we conceal the destructive changes made to executable memory (when it is read) from the adversaries. Heisenbyte allows legitimate read operations to disclose the contents of executable memory while keeping the randomized changes made to the read memory hidden. This enables us to transparently support existing COTS binaries without the need to ensure all legitimate data and code are separated cleanly and completely in the disassembly. The heart of Heisenbyte lies on the assumption that every byte in the executable memory can only be exclusively used as code or data.

Independent and concurrent to our work, Werner *et al.* also propose No-Execute-After-Read (NEAR) in a Destructive Code Read (DCR) implementation similar in spirit to Heisenbyte [168]. More recently, Pewny *et al.* combine the use of DCR and XoM in an implementation called Byte-

Granular DCR and XoM (BGDX) to reap the benefit of the legacy binary compatibility while being robust to code inference attacks [112]. We return to a comparison and discussion in Section § 4.7.

4.2.3 Assumptions

We assume a powerful adversary who can read (and write) arbitrary memory within the address space of the vulnerable program, and do so without crashing the program. On the target system, we also make similar assumptions used in related papers addressing the problem of memory disclosure attacks. We assume that the target system is equipped with the following protections:

- **W⊕X**: Memory pages cannot be both executable and writable at the same time. This prevents direct overwriting of existing code or injection of native code into the vulnerable program. We assume that this also applies to JIT code generated by programs, *i.e.* dynamically generated instructions cannot be executed on a memory page that is writable.
- **Load-time fine-grained ASLR**: All the static code from programs and libraries are loaded at random locations upon each startup. Address Space Layout Randomization (ASLR) reduces the predictability of the code layout. Furthermore, we require code layouts to be randomized at a fine granularity so that the registers [107] used and instruction locations within a function [76] or basic block [161] are different. Without this, an adversary can find code pointers in non-executable memory and infer the code layout of the rest of the memory without directly reading them.
- **Defenses against JIT attacks**: We also assume that fine-grained ASLR is applied to JIT engines [63], necessitating an adversary to perform a scan of the JIT memory pages to locate usable code reuse gadgets.

4.3 Heisenbyte Design

In this section, we describe our destructive code read primitive and how it thwarts memory disclosure attacks. Since our goal is to extend protection against memory disclosure attacks to

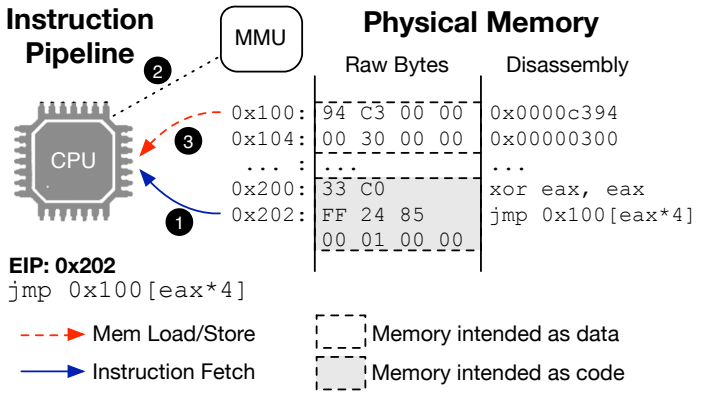

Figure 4.2: A typical execution of a `jmp` instruction using both code and data interleaved on the same memory page.

COTS binaries, we also detail the challenges in determining static data from code in disassembled binaries and how they motivate our defense approach.

4.3.1 Destructive Code Reads

4.3.1.1 Review of Instruction Pipeline

We briefly review what happens in the CPU pipeline when an instruction dereferences memory for its data. This is to familiarize the reader with the distinction between a memory read or write operation that uses memory as *data* and an instruction fetch operation (which is also a special form of memory read operation) that uses memory as *code*.

Figure 4.2 shows the execution of a `jmp` instruction, a typical implementation of a switch statement and a very common example of both code and data residing within the same memory page marked as executable. To aid explanation, we present the raw byte representation as well as its disassembled instructions. Without loss of generality, we assume the use of 4kB memory pages for the rest of our paper. While we have demarcated the bytes that are intended to be read as data from those intended to be executed as code, note that the processor is oblivious to this; all the processor knows of is the access permissions of a given memory page.

In Step ❶, the CPU performs a code fetch of the `jmp` instruction from the `0x202` address

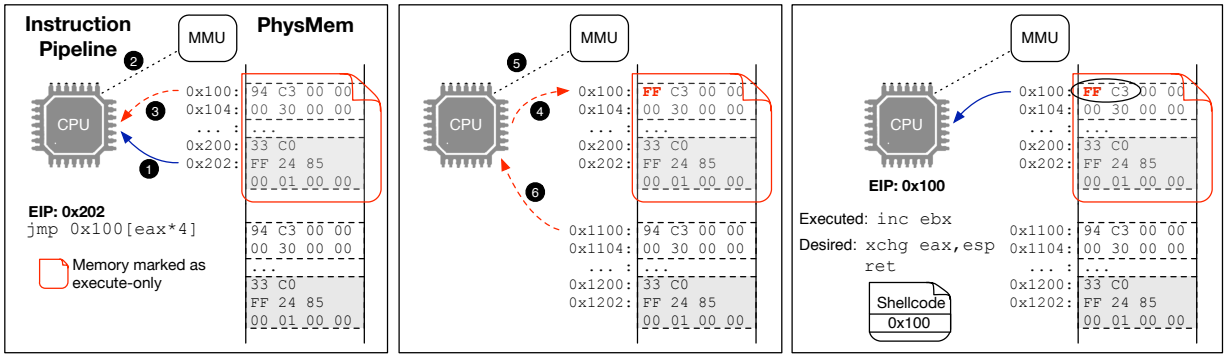

Figure 4.3: Destructive code read process.

pointed to by the Extended Instruction Pointer (EIP). The instruction is decoded and the CPU determines that it needs to dereference the memory at a base address of `0x100` and an offset given by the register `eax` for its branching destination. Since the address `0x100` is in the virtual addressing mode, the CPU has to translate the address to the corresponding physical address via the Memory Management Unit (MMU) in Step ❷. For simplicity, we assume an identity mapping of the virtual to physical addresses. Subsequently, the CPU dereferences the address `0x100` via a memory load operation in Step ❸, and completes the execution of the `jmp` instruction.

4.3.1.2 Destructive Code Read Process

In Figure 4.3, we detail the process of how destructive code read can thwart executable memory disclosure attacks. Every Windows program binary comes with a PE header that allows us to parse and identify all static memory sections that are marked as executable. We maintain a duplicate copy of these executable memory pages to be used as data in the event of a memory read dereferencing operation. Further, in order to detect read operations in the executable memory page, we need to mark that page as execute-only.

In Figure 4.3(a), we show this duplicate page directly below the executable page. Like in the earlier example, the instruction is fetched at Step ❶, and the memory address of the data to be dereferenced is translated via the MMU at Step ❷. When a memory dereferencing for the data

address occurs at Step ❸, this invokes a memory access violation.

The destructive code read begins at this point, shown in Figure 4.3(b). When we detect the read operation of the executable page, we overwrite the byte at the faulting memory address with a random byte at Step ❹. At Step ❺, via the MMU, we redirect the virtual address of the memory read to a different physical address that points to our duplicate page. We can then service the read operation transparently with the original data value at Step ❺, and the instruction that uses that data can function normally. Next, we show how these operations, specifically Step ❹, have set up a system state that can thwart a memory disclosure attack.

4.3.1.3 Thwarting Memory Scan Attacks

Since code and data are serviced by separate memory pages depending on the operation, the bytes that are read from executable memory pages may no longer be the same as the ones that can be executed at the same virtual address.

Given that a legitimate application has previously dereferenced the memory address `0x100` as data, the code memory address at `0x100` now contains a randomized byte. Executing the instruction at this address will lead to unintended operations. For instance, in Figure 4.3(c) if the adversary uses a memory disclosure bug to read the memory contents of `0x100`, she sees the *original* byte sequence "`94 C3`", which represents a commonly found stack pivot gadget[2]. Thinking that she has found the stack pivot gadget, she sets up her dynamic code reuse payload to use the address `0x100`. Since the earlier code read operation has "destroyed" the byte there with the random byte `FF`, when the code reuse payload executes the instruction at address `0x100`, the *garbled* byte sequence "`FF C3`" is executed as `inc ebx`. This effectively stems the further progress of the exploit.

[2]A sequence of instructions modifying the stack pointer to address a code location of the adversary's choosing

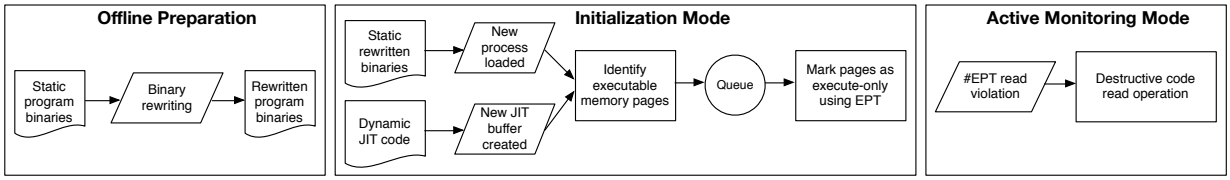

Figure 4.4: Flowchart of configuration of EPT for monitored executable pages.

4.3.2 Statically Separating Code and Data

Our use of destructive code reads in Heisenbyte at runtime is motivated by the (im)possibility of precisely and completely distinguishing disassembled bytes intended to be data from those intended to be instructions during runtime. This leads us to adopt a fundamentally different strategy from the earlier works that enforce execute-only memory using compiler-based techniques. Instead of determining the code or data nature of bytes during offline static analysis and enforcing runtime execute or read policies based on this, we infer the code/data nature of bytes at runtime, identify the inferred data bytes in executable memory, and remove the possibility of using them as executable code in attacks. We describe some of the main challenges of accurately identifying data in executable sections of Windows binaries, and how we sidestep these challenges using binary rewriting.

4.3.2.1 Challenges in Distinguishing Data from Code

Halting Problem Legitimate data must be separated out from the disassembled bytes of the executable sections of the binaries. To do so requires making a judgment on whether or not a range of bytes is intended to be used as data at runtime. While heuristics can be used to make that judgment, this code or data separation task at binary level essentially reduces to the halting problem because we can be sure only at runtime when bytes are truly intended to be code, and yet we want to do this during static analysis [162].

JIT Code Generation Web scripting languages such as Javascript are optimized for efficient execution by modern web browsers using just-in-time compilation. While the newer versions of

web browsers like Internet Explorer and Mozilla Firefox separate the code and data into different memory pages, with the latter in non-executable ones [9], the older versions however emit both code and data on same executable pages. We want to support the use of these legacy JIT engines.

Corner Cases In our analysis of Windows shared libraries, we found that there are many corner cases where the disassembler cannot accurately determine statically if a chunk of bytes is intended to be data or code. This stems from the limitations of the disassembly heuristics used by the disassembling engine.

A common example of incorrect disassembly is the misclassification of isolated data bytes as `RET` return instructions within a data block. A `RET` instruction is represented in assembly as a one-byte opcode, and can potentially be a target of computed branch instructions whose destination cannot be statically decidable. Therefore, the disassembler frequently misclassifies data bytes that match the opcode representation of return instructions as code.

We also found situations which assume that code and data sections are located in a specific layout. For example, in `kernel32.dll`, a shared library used by all Windows binaries, the relocation section indicates a chunk of bytes that are dereferenced as data at the base of the executable `.text` section. Because a readable and writable data section `.data` almost always follows this `.text` section, any instruction referencing this data also assumes that 400 bytes following this address has to be a writable location. This structural assumption is extremely difficult to discern during offline static analysis. If we blindly relocate this data from the executable `.text` section to another section without respecting this structural assumption, a crash is inevitable.

4.3.2.2 Our Conservative Separation Approach

As mentioned previously legacy COTS binaries, especially Windows native programs and libraries, have substantial amount of legitimate data interleaved with code in the executable sections. Blindly retaining these data can lead to exhorbitant overheads in Heisenbyte as read access to each of the these data items in the executable memory will incur the overhead of the destructive code read operation.

To mitigate these overheads, we perform very conservative static analysis to determine well-defined data structures that can be safely relocated out of the executable sections without affecting the functionality of the program. For instance, in many legacy Windows binaries, the read-only data sections are merged with the code section. This is not a problem because the format for the data section is well-documented. Similarly, we also handle well-structured data chunks like strings, jump tables and exception handling information. Here, we describe examples of these legitimate data chunks commonly interspersed with code in the executable sections of Windows COTS binaries.

Standard data sections Many Windows native binaries have the standard non-executable data-only sections embedded within the executable `.text` section. Examples include the Import Address Table, the Export Address Table and debug configuration section.

Merged data sections An optimization technique to minimize the file sizes of programs is to merge the read-only data section (`.rdata`) and the main executable section (`.text`)[3]. This technique is commonly used in Windows native binaries and shared DLL libraries. We are specifically targeting the relocation of two types of read-only data in this section, namely strings and Structured Exception handler (SEH) structures, since they are well defined.

Jump tables High-level switch statements are implemented as jump instructions and jump tables in assembly. Compilers typically position the jump table offsets near the jump instructions that use jump tables. These jump tables are intended to be dereferenced as data at runtime.

4.4 System Implementation

In this section, we detail the various components of Heisenbyte, and how we realize the mechanism of destructive code reads on selected executable memory pages. As shown in Figure 4.4, we achieve this in three different stages. We begin by rewriting the program binaries that we want to protect to separate specific data from the code in an **Offline Preparation** stage. We detail this

[3]This can be achieved using Microsoft Visual Studio compiler with the linker flag `/merge:.rdata=.text`.

process in § 4.4.1.

To ensure that our destructive read operations only apply to the processes we want to protect, Heisenbyte processes targeted executable memory pages in the following two modes. We discuss each of them in detail in § 4.4.2.

- **Initialization mode**: This mode identifies at runtime selected executable memory pages to protect, and subsequently configures execute-only access permissions for these pages, in preparation for the next mode.
- **Active monitoring mode**: Once the set of executable pages is configured with the desired EPT permissions, this mode is then responsible for performing the destructive code read operation when it detects a read operation to an executable page.

Furthermore, to demonstrate that the technique is practical on COTS binaries, we invest substantial effort in this work to develop Heisenbyte to work on the primarily close-sourced Windows OS. The techniques and design presented in this work can be generalized to other OSes like Linux.

4.4.1 Offline Static Binary Rewriting

Recognizing well-defined data in disassembly We use the state-of-the-art commercial disassembler, IDA Pro, to generate the disassembled code listing of the programs. We also leverage IDA Pro's built-in functionality to identify well-defined data structures (described in earlier sections) commonly found in executable memory pages.

Rewriting engine We develop our binary rewriting engine as a Python script. Unlike traditional binary rewriting tools, we do not perform any rewriting operations that change the semantics of instructions. Our engine focuses on using disassembly information from IDA Pro and the section headers to determine if a range of bytes within an executable section needs to be relocated to a separate data section. Our engine reconstructs the PE header to add a new non-executable section to consolidate all these identified data. Relocation information is crucial

in aiding both our static analysis and our relocation operations. For example, if a range of data bytes needs to be relocated to another section, the relocation table is updated either by adding new relocation entries or editing existing ones to reflect the new location of the relocated data. Relying on the relocation tables allows us to transparently move bytes around within a PE file without breaking the functionality of the program.

Overcoming Windows binary protection To evaluate our rewritten Windows native library files with Heisenbyte, we need to replace the original files. However, on Windows, critical shared libraries and program binaries are protected by a mechanism called Windows Resource Protection (WRP) [99]. WRP prevents unauthorized modification of essential library files, folders and registry entries by configuring the Access Control Lists (ACLs) for these protected resources. Only the Windows Installer service, *TrustedInstaller*, has full permissions to these resources.

To get around this problem, we rely on the fact that we have administrative privileges on the system. We take control of the ownership of the protected files from the *TrustedInstaller* account using the command `takeown.exe`, and grant to our account full access rights for the protected files using `icacls.exe`. At this point, we can rename the files but we cannot replace the files because they are still in use. We rename the files and copy our rewritten binaries with the original filename. When the system is rebooted, our rewritten libraries will be then loaded into the system. To ensure integrity of the binaries, the modified ACLs of the protected binaries are restored after the rewritten binaries are replaced.

This technique of deploying rewritten Windows native files work for most of the binaries with one exception – `ntdll.dll`. The integrity of this file is verified when the system starts up. We solve this by disabling the boot-time integrity in the bootloader [49], so that the rewritten `ntdll.dll` binary can be loaded.

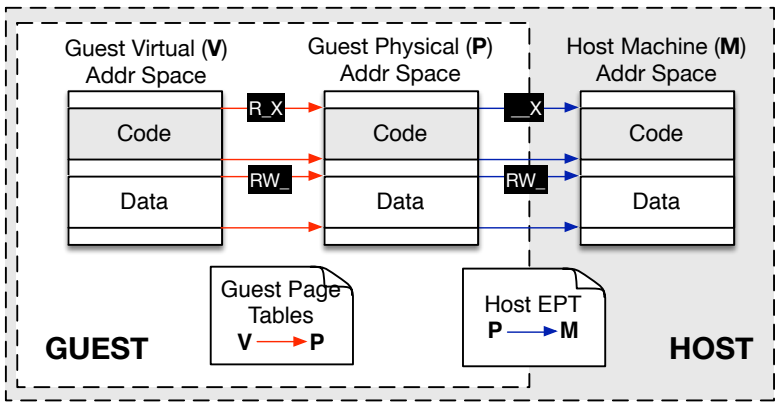

Figure 4.5: Nested paging structure using virtualization hardware support (using Intel-specific terms).

4.4.2 Heisenbyte Core Monitoring Components

4.4.2.1 Review of Intel Extended Page Tables (EPT)

Before we discuss each of the components in the two modes, we first describe the key hardware virtualization feature we use to achieve our goals.

Heisenbyte needs to be able to detect when executable memory is being read. There are a number of ways to do this: mediating at the page fault handler [11] or leveraging the split-TLB microarchitecture of systems [51]. These solutions stem from the limitation of current OSes not being able to enforce execute-only permissions on memory pages. Fortunately, hardware virtualization support – hardware-assisted nested paging – on commodity processors provides a means for us to enforce fine-grained execute-only permissions on memory pages. This hardware feature augments existing page walking hardware with the ability to traverse in hardware the paging structures mapping guest physical (P) to host machine (M) addresses. This eliminates the overhead involved in maintaining shadow page tables using software. A virtualization-enabled MMU maps virtual (V) addresses in the guest to machine physical addresses in the host, using both the guest page tables and the host second-level page tables[4]. This is done transparently of the guest OS.

[4]Intel terms this Extended Page Tables(EPT), and AMD calls this Nested Page Tables (NPT)

We show three address spaces spanning across the guest and host modes in Figure 4.5. In the guest, the page tables store the V→P address mappings, as well as the corresponding permission bits. These guest page tables, described earlier, cannot be configured with solely the execute bit set. Conversely, in the host, the EPTs maintain the P→M address mappings. The key difference between the EPTs and guest page tables is that the EPTs can configure each page mapping as execute-only. When an access to a memory page violates the permissions configured for that page, an #EPT violation is invoked, transferring control to the hypervisor.

This mechanism is instrumental in our system to detect read operations to executable memory. In our work, like Readactor [33], we rely on hardware-assisted EPT to configure guest physical memory pages as execute-only with no read or write access. Since this is a virtualization-assisted technology, virtualization has to be enabled on the system we are trying to protect. On systems that need to protect existing virtualized guests, Heisenbyte can be implemented within the Virtual Machine Monitor (VMM) software, such as Xen or KVM. However, the need for virtualization does not preclude the protection of non-virtualized systems.

To demonstrate this, we make a conscientious effort to implement Heisenbyte for a non-virtualized OS. We develop Heisenbyte as a Windows driver that will configure the EPT paging structures, enable virtualization mode and place the execution of the non-virtualized OS into virtualized guest mode (non-root VMX mode). Heisenbyte does this on a live running system, without requiring any system reboot. The *host mode component* (shown in Figure 4.6) of our driver ensures that the running system functions as usual, by configuring the EPT structures to use identity mappings from the guest physical to host machine addresses. At this point, our host mode component is in a position to configure the execute-only permissions transparently of the guest OS.

4.4.2.2 Identifying Executable Memory

Before we can configure the EPT execute-only permissions, we need to first identify which executable memory pages to monitor. To do that, we have to track when and where executable

memory from processes are loaded and mapped. Since the treatment of dynamic code tracking is more involved, we will describe them in detail separately.

Static program binaries To deal with static code, Heisenbyte *guest mode component* (as shown in Figure 4.6) begins its initialization by registering Windows kernel-provided callback functions associated with the creation/exiting of processes and loading/unloading of shared libraries. Using the callback registration APIs, `PsSetCreateProcessNotifyRoutine` and `PsSetLoadImageNotify`, our driver guest component is informed whenever a new static code process or library gets loaded. This callback mechanism applies to both executable files and shared library files. If a newly loaded static image matches within a whitelist of binaries we are protecting, our guest mode component parses the memory-mapped PE header to get the list of guest virtual addresses and sizes of the executable sections in each loaded image.

With the guest virtual addresses, we need to retrieve the corresponding guest page table and guest physical addresses for each virtual memory page to configure the EPT entries. However, since the OS performs a lazy allocation when doing the memory mapping, these memory pages may not be paged into memory yet. As a workaround, Heisenbyte schedules a thread within the context of the target process and accesses one byte in each memory page to invoke the paging-in mechanism. Further, Heisenbyte uses the `MmProbeAndLockPages` kernel API to make the pages resident in the physical memory, so that they cannot be paged out. This necessarily increases the memory working set of a program. We will investigate this in § 4.5.2.2.

These information is stored in a queue buffer shared by the guest mode and host mode components. It is noteworthy that since the guest mode component runs in the VMX non-root guest mode, it has no access to the EPTs. The configuration of the EPT mappings has to be performed by the host mode component.

Dynamic JIT code Unlike the loading of static binaries into memory, dynamic memory buffer creation/freeing does not have convenient kernel-provided callbacks. Furthermore, the protection bits of a dynamic buffer may change at runtime during the generation and execution of dynamic code. For example, a modern JIT-enabled browser, like Safari, first allocates a

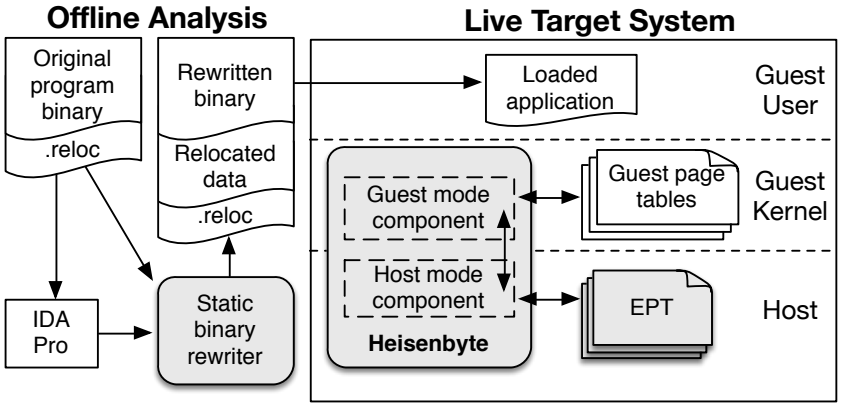

Figure 4.6: Overview of system architecture (Heisenbyte components are shaded grey).

writable (read/write RW) buffer as a code cache to fill with generated native code. With our assumption that hardware W⊕X DEP is enforced, the JIT engine has to remove the writable permission and make the code cache executable (read/execute RX) before executing the code cache. If the dynamic code cache subsequently needs to be modified, the buffer is restored to a writable (read/write RW) one before changes to the code cache can be made.

Based on the lifetime of the buffer during which the code is ready to be executed, we observe that we only need to monitor the buffer during this period of time. Specifically, we begin tracking a dynamic buffer when the protection bits changes from non-executable to executable, and stop tracking a dynamic executable buffer when it is freed or when its executable bit is removed.

Windows-specific implementation Next we discuss how we detect when dynamic memory buffers are turned executable and when they are freed. All operations that are used to free or change protection bits of memory result in two functions in `ntdll.dll`, `NtFreeVirtualMemory`, and `NtProtectVirtualMemory` respectively, just before invoking the system calls to the kernel services. When `ntdll.dll` is loaded into our target process, we modify the entry points of these two functions with trampolines to a Virtual Memory (VM)-tracking code that resides on a dynamically allocated page. Since the function hooking is performed in-memory, the OS Copy-on-Write mechanism ensures that these hooks only apply to the target process.

In practice, dynamic memory buffers are created and freed very frequently. Since we are only

interested in executable buffers, we use an auxiliary bitmap data page to indicate if an executable buffer of a given virtual address has been previously tracked. This added optimization enables the VM-tracking code to decide if it should handle specific events.

The VM-tracking code that monitors the changing of protection bits of buffers performs a hypercall to our host mode component whenever an executable buffer is configured to be non-executable and vice versa. The host mode component updates the address bitmap depending on whether a new executable page is being tracked or removed from tracking. Conversely, the VM-tracking code that monitors the freeing of executable buffers will perform a hypercall when it determines from the bitmap that a buffer with a given virtual address is being freed. The host mode component will then reset the EPT mapping for the physical pages of the buffer to an identity mapping, essentially stopping the tracking of this dynamic executable buffer.

Protecting VM-tracking code and data The VM-tracking code resides on a dynamically allocated executable page, and is protected by Heisenbyte just like any typical executable memory page. Conversely, by being configured to be read-only from the userspace, the auxiliary bitmap is protected from any tampering attacks originating from the userspace; it can only be modified in the host kernel mode (specifically by the host mode driver component. Furthermore, a XOR-based checksum of the bitmap is maintained and verified before the bitmap is updated in the host mode component.

4.4.2.3 Overcoming Challenges in using EPT

Problem of shared physical memory pages One key challenge in using EPT to enforce execute-only memory is that the guest physical memory pages may be shared by multiple processes due to the OS's Copy-on-Write (COW) optimization. This COW mechanism is a common OS optimization applied to static binaries to conserve physical memory and make the startup of programs faster. Thus the OS lazily duplicates the original page into a newly allocated physical page only when the process writes to the memory page. Before these physical memory pages are duplicated by COW, they are shared by multiple processes. Enforcing execute-only permissions

on these shared guest physical pages may result in many #EPT violations triggered by processes we do not care about and cause unnecessary overhead.

Inducing COW on physical pages Heisenbyte overcomes this problem by inducing COW on the executable memory pages of target processes. We leverage the guest OSes' innate COW capability to transparently allocate new physical memory pages for the static code regions of processes we want to protect. To invoke COW on the memory pages of processes, the write operation must occur in the context of the process; a write operation originating from the hypervisor into the memory space of a user process will not trigger the copy-on-write mechanism.

When a static binary is loaded into memory, Heisenbyte schedules an Asynchronous Procedure Call thread [98] to execute in the context of the target process. This thread suspends the execution of the original target process, enumerates the static code regions of the process using the PE headers mapped in the address space, and performs a read and write operation on each executable memory page. This identity-write operation is very efficient since we only "touch" one byte in each 4kB memory page. The OS detects this memory write and invokes the COW mechanism. In this manner, each executable static page in a process will no longer share a physical page with another process.

The executable memory pages are then configured to be read-only using EPT by the host mode component only after the COW-inducing thread has completed processing all the executable memory pages of the newly loaded binary.

4.4.2.4 Intervention with Code Garbling

Maintaining separate code views To enable our destructive code read operations while allowing legitimate data reads in executable memory to function properly, we need to maintain separate code and data views for each executable memory page we are protecting. We leverage the EPT to transparently redirect the use of any guest virtual address to the desired view at runtime. In Figure 4.7(a), before a target process is being protected, an identity EPT mapping of the guest physical to host machine memory is maintained.

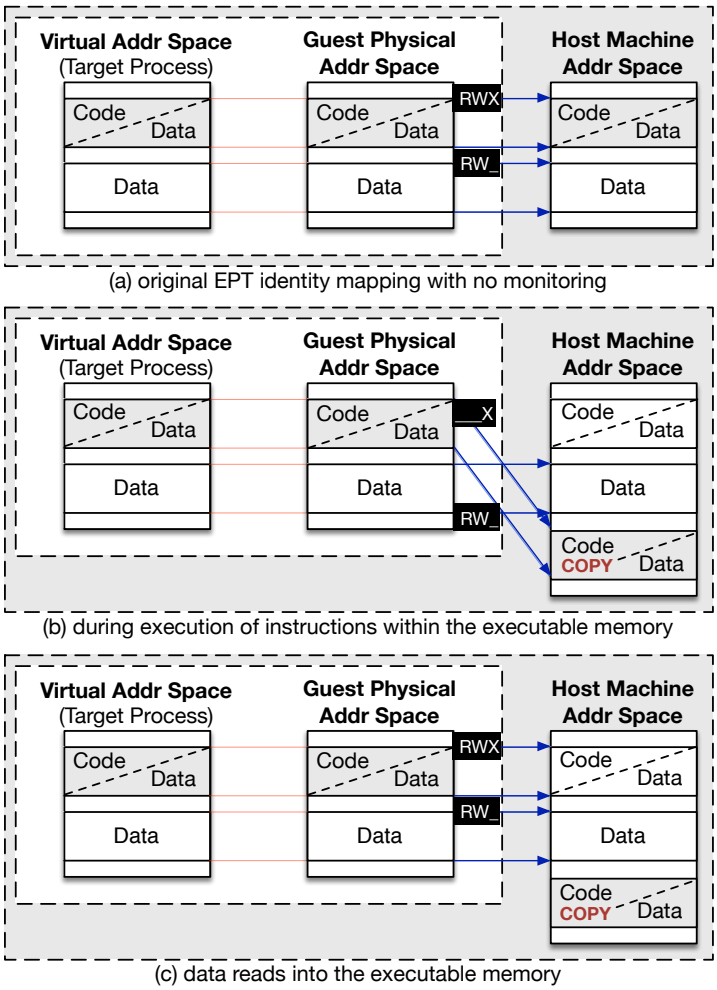

Figure 4.7: Using EPT to maintain separate code and data views transparently.

After identifying the guest physical memory pages to protect, we add a duplicate page in the host machine address space. Any subsequent instructions being executed are redirected to the *code copy* memory page shown at the bottom of Figure 4.7(b). This guest physical page is configured to be execute-only using EPT.

Destructive reads into executable memory With the executable pages configured to trigger a VM exit upon a data read, our #EPT violation handler in the host mode component of the driver can intervene and mediate at these events. At each #EPT read violation, we overwrite the data read address within our *code copy* page with a random byte. This constitutes the destructive nature of our code reads. Since there are legitimate data reads into executable memory from the

kernel, especially during PE loading, we perform the byte garbling only when the read operation originates from user-space.

Next we edit the EPT entry to have read/write/execute access and redirect the read operation to read from the original code page, now intended exclusively to service data read requests, as shown in Figure 4.7(c). To restore the memory protection, we set the single-step trap flag in the EFLAGS so that a VM exit is triggered immediately after the instruction performing the read operation. At this point, we restore the EPT permissions to execute-only to resume operation.

4.5 Evaluation

In this section, we demonstrate the utility of Heisenbyte in stopping attacks that use static and dynamic memory disclosure bugs. We evaluate the performance and memory overhead of our system. Our experiments are done on 32-bit Windows 7 running on a quad-core Intel i7 processor with 2GB RAM. As our prototype does not handle SMP systems, we configure the system to use only one physical core.

4.5.1 Security Effectiveness

4.5.1.1 Memory Disclosure Attack on Static Code

We use the Internet Explorer (IE) 9 memory disclosure vulnerability (CVE-2013-2551) presented by Snow *et al.* [133]. This is a fairly powerful heap overwrite vulnerability involving a Javascript string object. It enables an adversary to perform arbitrary memory read and write operations repeatedly without causing IE to crash. On our test setup, we craft an exploit that leverages this memory disclosure bug as a memory read and write primitive.

As ASLR is enabled by default – Window's ASLR is a coarse-grained form that changes only the base addresses of the shared libraries at load time –, the exploit has to look for suitable code reuse "gadgets" to string together as an attack payload. To demonstrate that our system

works with an exploit that uses disclosed executable memory contents, we craft our exploit to dynamically locate a stack pivot ROP gadget.

The exploit begins by first leaking the virtual table pointer associated with the vulnerable heap object. This pointer contains an address in the code page of `VGX.dll` shared library. Using the memory read primitive, the exploit scans backwards in memory for the PE magic signature `MZ` to search for the PE header of the shared library.

It is noteworthy that at this point, if IE uses any code within the range of bytes the exploit has scanned, IE will crash due to the corruption of legitimate code by the destructive code reads. However, in a real deployment, as defenders, we do not want to rely on such opportunistic crashes. We assume that the exploit avoids scanning executable memory during this stage and only reads non-executable memory.

When the exploit finds the PE header of the library, it can then derive the base address of `user32.dll` by parsing the import address table in the PE header. The shared library `user32.dll` contains a set of ROP gadgets that are found offline. With this, the exploit can construct its ROP payload by adjusting the return addresses of the pre-determined ROP gadgets with the base address of `user32.dll`. To simulate the dynamic discovery of "gadgets" in a dynamic code reuse exploit, we craft the exploit to perform a 4-byte memory scan at the location of the stack pivot gadget, and then redirect execution to that stack pivot gadget.

While our actual system uses a randomized byte to garble the code, we use a fixed `0xCC` byte (*i.e.* a debug trap) for the code corruption in this experiment. This allows us to be sure that any crash is directly caused by our destructive code reads. When control flow is redirected to the stack pivot gadget, IE crashes at the address of the stack pivot with a debug trap. This demonstrates that Heisenbyte stems the further progress of the exploit as a result of corrupted byte caused by the exploit's executable memory read.

Furthermore, we configure the `Windbg` debugger to automatically launch upon application crash. When the debugger is invoked at the crash address at the location of the stack pivot, the debugger displays and disassembles the original byte sequence of the stack pivot gadget in

`user32.dll`. As the debugger reads memory as data read operations, the original bytes at that code address are shown. It is apparent that what gets *executed* is different from what gets *read*[5]. This further demonstrates that Heisenbyte correctly maintains separate code and data views of executable memory.

4.5.1.2 Memory Disclosure Attack on Dynamic Code

At the time of writing, we are aware of only one publicly available exploit [113] that uses an integer overflow bug to achieve memory read/write capability on the JIT code cache of mobile Chrome. However, this exploit only works on ARM devices, so we cannot use this for our evaluation.

To evaluate our system on memory disclosure attack on dynamically generated code, we create a vulnerable program that mimics the behavior of JIT engine in the creation of dynamic executable buffers. Our program allocates a readable and writable buffer and copies into this buffer a pre-compiled set of instructions that uses a jump table. This is similar to the behavior of legacy JIT engines that emit native code containing both code and data in the dynamic buffer.

With the code cache ready to execute, our program makes the dynamic buffer executable by changing the permission access to readable/executable, and executes the buffer from the base address of the buffer. The program functions correctly with Heisenbyte running. Since the jump tables in the dynamic buffer are only ever used as data in the lifetime of the buffer, Heisenbyte properly supports the normal functionality of the simulated JIT-ed code.

To simulate an attack that scans the memory of the dynamic code region for code reuse gadgets, we create an exploit to leverage a memory disclosure bug we have designed into the program. The exploit uses this bug to read the first four bytes of the dynamic buffer and redirects execution control to the start of the dynamic buffer. Like in the case of the experiment with IE9, the vulnerable program crashes at the base address of the dynamic buffer as a result of the destructive code reads induced by Heisenbyte.

[5]This incidentally can be a real pain for someone trying to debug the code crash problem with a debugger.

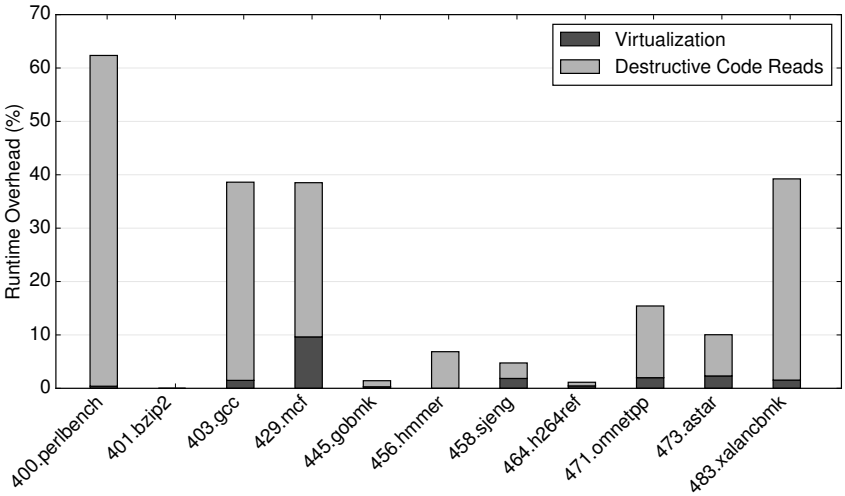

Figure 4.8: SPEC2006 execution overhead.

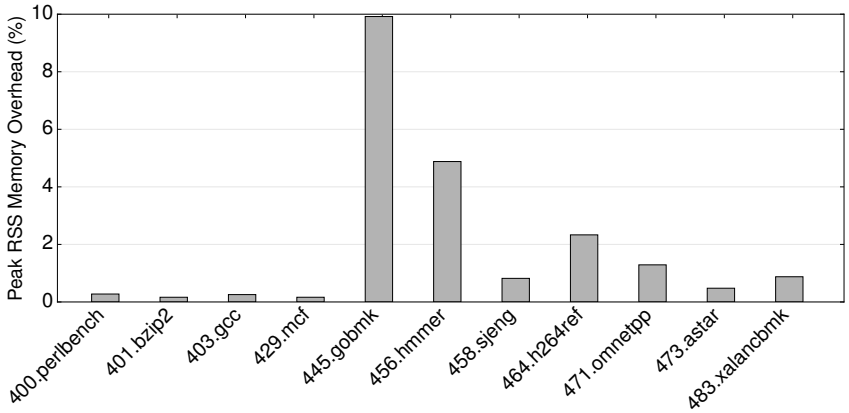

Figure 4.9: Memory overhead in terms of peak RSS.

4.5.2 Performance Overhead

4.5.2.1 Execution Overhead

We measure the slowdown caused by various components of Heisenbyte using the SPEC2006 integer benchmark programs. Since our solution works on and rewrites binaries, we first compile the programs and work with the compiled binaries assuming no source code is available. We compile the SPEC2006 programs with Microsoft Visual Studio 2010 compiler using the default linker and compilation flags. As the compiler does not support the C99 feature, *e.g.* type `_complex`, we

cannot successfully compile `462.libquantum`. We thus use only 11 out of 12 SPEC2006 integer applications for our evaluation. For all the tests, we restart each set of runs on a rebooted system, perform 3 iterations using the base reference input and take the median measurements.

We evaluate the execution slowdown caused by Heisenbyte to an originally non-virtualized system. The overhead of Heisenbyte comprise two main sources, namely the overhead as a result of virtualizing the entire system at runtime, and the overhead of incurring two VM exits for each destructive code read operation. Separating the measurements for the two allows us to evaluate the overhead net of virtualization when Heisenbyte is deployed on existing virtualized systems (they are already occurring the virtualization overhead).

To measure the overhead caused by purely virtualizing the system, we run the SPEC benchmarks with the Heisenbyte driver loaded, but without protecting any binaries or shared libraries. Compared to a baseline system, the virtualization overhead ranges from 0% (`401.bzip2`) to 9.6% (`429.mcf`). The virtualization overhead is highly dependent on the execution profile of the programs. We attribute the high overhead for `401.bzip2` to the paging operations performed by Intel EPT hardware page walker. On average, the geometric mean of the virtualization overhead caused by Heisenbyte is 1.8% across all the programs.

With the measurements for the virtualization overhead, we can now measure the overhead of the destructive code reads due to the incomplete removal of legitimate data from the executable memory pages. We configure Heisenbyte to protect the SPEC binaries and all the shared DLL libraries used by SPEC, and compare the execution time to the baseline. The variance in this overhead is huge, depending on how much legitimate data is not removed by the binary rewriting. The destructive code read overhead ranges from 0% (`401.bzip2`) to 62% (`400.perlbench`), with an average of 16.5% across the programs. This overhead is a direct consequence of the imperfect removal of legitimate data from the executable memory pages at the binary rewriting stage. The higher the frequency a program accesses such legitimate data in the memory pages, the greater the overhead incurred by the destructive codes. The average of the combined virtualization and destructive code read overhead is 18.3%.

In this work, we choose to be very conservative in the types of data that we relocate out of the executable sections during the binary rewriting to show that the system can still tolerate the incomplete relocation of *all* data from the executable sections. This overhead can be further reduced with a more aggressive strategy in removing the data.

4.5.2.2 Resident Memory Overhead

As discussed in § 4.4.2.2, Heisenbyte requires keeping the executable memory pages resident in physical memory when configuring the EPT permissions and monitoring for data reads to these pages. Here we evaluate how much more physical memory overhead introducing Heisenbyte causes. We measure this by tracking the *peak* Resident set size (RSS) of a process over entire program execution. RSS measures the size of process memory that remains resident in the RAM or physical memory. We inject a profiling thread to our processes to log the current maximum RSS as the process runs every 20 seconds. Figure 4.9 shows a modest increase of 0.8% on average in the peak RSS across all the programs.

4.6 Related Work and Enhancements

Our work is enabled by two key techniques, namely the ability to maintain separate code and data views in a von Neumann memory architecture[6], and destructive read operations applied on executable memory. We have described the research works most closely related to our work in § 4.2.2. Here we detail other works using the above two techniques. Then we discuss possible enhancements to Heisenbyte.

Hardware-based destructive reads Examples of destructive read operations in practice are sparse. The destructive-read embedded DRAM [41] is a special-purpose DRAM that allows destructive reads to conserve power consumption. The contents of the memory can only be read once. At the software level, destructive read operations are sometimes performed by the BIOS

[6]where code and data are stored in the same addressable memory

during the memory check in its Power-On Self Test (POST), with the purpose of ensuring sensitive memory contents cannot be leaked [58]. Our software-emulated destructive read primitive on executable memory represents the first work (together with an independent and concurrent work, NEAR [168]) to apply this technique to make system states non-deterministic and harder for adversaries to leverage code memory disclosure vulnerabilities.

Maintaining separate code/data views Many have explored the value of maintaining separate views for code and data. The earliest works are mostly offensive in nature. Van Oorschot *et al.* leverage the process of desynchronization the TLB to bypass self-hashing software checks [155]. Shadow Walker, a rootkit, relies on the split-TLB architecture of processors to hide its malicious code from being detected by code scans by Antivirus [137]. Torrey explores the use of EPT to differentiate code from data at runtime to perform attestation on dynamically changing applications [151]. Spider also uses EPT permissions to maintain different views for code and data to implement the evasion-resistant breakpoints that are "invisible" to the guest [38]. Our work shares similar EPT-based techniques with these works, albeit towards different goals.

New hardware features to reduce overhead In this work, we choose to implement Heisenbyte with the standard virtualization features found in most processors. The goal is to provide a baseline proof-of-concept implementation of our design. As we have seen in § 4.5.2, the major source of overhead comes from inducing the VM exits to implement the destructive code reads. This can be reduced substantially with the combined use of two new virtualization features in the recent Haswell processor [68]. This processor allows selected `#EPT` violations to be converted to a new type of exception that does not require VM exits to the hypervisor. The latency of VM exits can then be reduced substantially. This exception is known as the `#VE` Virtualization Exception. With this feature, during the active monitoring mode, a data read into protected executable memory pages will trigger an exception and control will be handed over to the guest OS `#VE` Interrupt Service Handler (ISR). To handle the configuration of EPT entries, the second feature, named EPT Pointer switching, allows the guest OS to efficiently select within a pre-configured set of EPT pointers having the required EPT permissions we need.

Graceful remediation In additional to detecting attacks, Heisenbyte can offer the capability to gracefully terminate, instead of crashing, the process that is being targeted by the attack, and provide further alerting information regarding the attack to the user. Instead of using randomized junk bytes for the destructive code reads, Heisenbyte can use specific bytes designated to induce selected software interrupts or traps when executed. The host component of Heisenbyte can be configured to mediate on these interrupts. When malicious code attempts to execute code modified by earlier reads, pertinent information about the attempted code execution, such as the faulting instruction, and the original and modified contents of the executable memory page, can then be logged. This may assist in identifying the associated vulnerability, and provide useful forensics information for vendors to patch the program.

Size of garbled code At present, Heisenbyte disregards the operand size of the instruction performing the reads into the executable memory, and performs destructive code reads of only one byte. An adversary who uses data reads of four bytes to scan the memory can potentially exploit this. Garbling only one byte will give the adversary the potential to use the remaining three bytes from the data reads. To tackle this problem, Heisenbyte can easily be extended to handle code reads using different operand sizes. We can maintain three hashtables, each storing the opcodes used for 1-byte, 2-byte and 4-byte operands. Whenever a code read happens, Heisenbyte can look up the hashtable to determine efficiently the size of operand and destroy the same number of bytes accordingly.

Code read logs to guide binary rewriting As an optimization to aid the offline static analysis, we can augment Heisenbyte to record all read operations into executable memory into a log buffer. This dynamic log can be used to build a "bitmap" of sorts, similar to that used in BGDX [112], to indicate definitively at runtime which bytes are code and data, after which the binaries can be analyzed and rewritten repeatedly using this information to achieve a high code coverage over time. This can further reduce the overhead of the system, since the data reads that previously trigger VM exits will no longer occur.

4.7 Discussion

After Heisenbyte [147] was published as a binary-compatible defense against dynamic code reuse attacks, further research highlighted its weaknesses by formalizing generic properties that destructive code reads-based defenses should hold to be secure. Several works subsequently hardened destructive code reads from attacks that undermine these properties. In this section, we describe these works and reflect upon this line of research efforts to defend against memory disclosure-based dynamic code reuse attacks.

Execute-Only Memory (XoM) By reducing the information available to attackers, XoM works well against exploitation through memory disclosure. Its value is evident as we see XoM making its way into commodity hardware such as the newer ARMv8 processors[7] and Intel processors in the form of Intel Memory Protection Keys[8]. For hardware that have yet to support this feature, researchers have developed software-based solutions to approximate the defensive principle of XoM, with hardware virtualization support [33, 32], hooking the page fault handler [11] or novel use of the split-TLB [51]. These XoM-based defenses require code and data to be cleanly separated, and thus work well only when source code is available for recompilation or when the program binaries can be accurately disassembled when compiled with specific compilers like gcc and clang [8].

Binary-Compatibility of Destructive Code Reads While Andriesse *et al.* show that there is no interspersed code and data in programs compiled with modern `GCC` and `clang` compilers [8], recent work have highlighted that this problem still exists on both Windows and ARM platforms, especially for programs compiled with Microsoft Visual Studio compiler [168, 112]. Destructive code reads (DCRs) is first conceived precisely to address this limitation, *i.e.* to protect legacy COTS binaries from code disclosure. Consequently, the primary source of overhead comes from protecting segments of programs where one is not confident of discerning a priori whether specific bytes are code or data. This is the main reason why Heisenbyte is designed with a best-effort,

[7]The ARMv8 architecture allows execute-only user permissions by clearing the `PTE_UXN` and `PTE_USER` bits.

[8]Execute-only permissions can be configured for memory pages using the Intel `RDPKRU/WRPKRU` instructions.

albeit very conservative, static analysis phase to distinguish between code and data definitively and then rewriting these binaries to lower DCR-driven overheads.

Therefore, a promising avenue to reduce the overhead of DCRs lies in being able to discern code from data as confidently and as much as possible during the static disassembly of programs. Towards this end, NEAR [168] achieves lower SPEC2006 overheads (5.72%) than Heisenbyte (16.48%) due to better heuristics (such as identifying jump tables and function-local data) in the statically identifying data within the code segments. Subsequently, BGDX [112] further improves upon the DCRs in two main aspects: (1) it achieves better coverage and accuracy in separating code from data by leveraging dynamic profling (DynamoRIO), and (2) it combines DCR with XoM on memory bytes that it has determined to be definitively code. These allows BGDX to achieve lower overheads (3.95%) and byte-granular memory protection.

Explicit vs Implicit Code Disclosure Attacks The initial implementations of Destructive Code Reads, both Heisenbyte [147] and NEAR [168], are designed to guard against *explicit* code disclosure attacks, *i.e.* disclosing the contents of executable memory by directly reading it. However, apart from explicitly reading memory, contents of memory can also be disclosed *implicitly* either by side channels attacks or code inference attacks.

Timing-based side channel attacks have been shown to be able to reveal memory bytes without directly reading them [127, 44]. Comprehensive protection against side channel leaks is generally recognized as a prohibitively challenging tasks in the general context, and not just pertaining to the disclosure of executable memory. Most of these attacks require programs to be crash-resilient. Most Windows COTS programs are not tolerant of crashes. Furthermore, exploiting these user applications is time-sensitive. For example, an attacker loses the opportunity to exploit the system once its exploit invokes a crash on IE or takes too long. These aforementioned reasons make existing side channel-based memory disclosure attacks on this class of binaries challenging.

Code inference attacks prey on the imperfections of the type of code randomization deployed together with DCRs. To inform the susceptibility of DCRs to code inference attacks, Snow *et al.*

formalize three generic properties (code *persistence*, *singularity* and *dis-association*) that DCRs must exhibit to remain secure [134][Section III.B]. Specifically, code must not have predictable instruction-level structure (dis-association), or possess temporal (persistence) and spatial (singularity) copies. If so, especially in target applications with scripting capabilities, DCRs can be undermined and usable code bytes can be inferred without ever reading the executable memory directly [134]. This largely stems from the narrow-scoped code transformations limited by binary-only code randomization techniques. Furthermore, Pewny *et al.* shows that even when coupling DCRs with extremely fine-grained per-load randomization, DCRs can remain vulnerable to code inferences by fingerprinting "code anchors" and function call-sites [112]. These attacks underscore the deployment challenges of DCRs in practice – the efficacy of DCRs is contingent on the type and scope of code randomization being deployed.

These attacks exploit predictable patterns in the executable memory contents. Subsequent countermeasures harden DCRs from such attacks by improving the type and scope of code randomization being deployed. To break temporal predictability of code memory, instead of merely using load-time randomization, dynamic continous binary-compatible code randomization schemes like Shuffler [170] and TASR [19] can be deployed to make the attack knowledge gleaned from code disclosure attacks transient. Furthermore, the range and scope of byte "garbling" in DCRs can be extended to break the predictability of instruction-level structure. To deal with code inference attacks, Morton *et al.* propose a DCR design where the portion of code that is destroyed includes not just bytes that are involved in memory read operations, but also all the other instructions/memory bytes that could potentially be inferred [103].

4.8 Conclusions

We present the novel formulation of destructive code reads to restrict adversaries' ability to leverage executable memory that are exposed using memory disclosure bugs as part of an attack. Commodity hardware feature is instrumental to the realization of this technique. We repurpose

commodity hardware virtualization support to provide timely and efficient mediation of read operations on executable memory. Heisenbyte guarantees the disclosed executable memory cannot be executed as intended, while still tolerating some degree of data not removed from the code pages. Our experiments demonstrate that Heisenbyte prevents the use of disclosed executable memory in real and synthetic attacks, while offering transparent protection for legacy close-sourced binaries, at modest overall runtime overheads.

Moving target principles like randomization in various forms raise the bar for exploitation; other forms of reducing or eliminating knowledge exploitable by attackers are necessary. With the conception of destructive code reads in Heisenbyte, we show that destroying information is yet another principle to remove this knowledge. Notwithstanding its limitation, destructive code reads remain an effective binary-compatible primitive in the gamut of randomization-based defenses when employed carefully in concert with both execute-only memory and code randomization schemes.

5

HADES: Detecting Malware with Microarchitectural Profiling

Assistive debugging hardware features facilitate efficient auditing of such microarchitectural events in a program-transparent manner.

Hardware-software interaction can be modeled as microarchitectural events to detect anomalous malicious code execution.

Recent works have shown promise in detecting malware programs based on their dynamic microarchitectural execution patterns. Compared to higher-level features like OS and application observables, these microarchitectural features are efficient to audit and harder for adversaries to control directly in evasion attacks. These data can be collected at low overheads using widely available hardware performance counters (HPC) in modern processors. In this chapter, we describe how we advance the use of hardware supported lower-level features to detecting malware exploitation in an anomaly-based detector. This allows us to detect a wider range of malware, even zero days. As we show empirically, the microarchitectural characteristics of benign programs are noisy, and the deviations exhibited by malware exploits are minute. We demonstrate that with careful selection and extraction of the features combined with unsupervised machine learning, we can build baseline models of benign program execution and use these profiles to detect deviations that occur as a result of malware exploitation. We show that detection of real-world exploitation of popular programs such as IE and Adobe PDF Reader on a Windows/x86 platform works well in practice in a prototype we called Hades [145]. We also examine the limits

and challenges in implementing this approach in face of a sophisticated adversary attempting to evade anomaly-based detection. The proposed detector is complementary to previously proposed signature-based detectors and can be used together to improve security.

5.1 Introduction

Malware infections have plagued organizations and users for years, and are growing stealthier and increasing in number by the day. In response to this trend, defenders have created commercial antivirus (AV) protections, and are actively researching better ways to detect malware. An emerging and promising approach to detect malware is to build detectors in hardware [37]. The idea is to use information easily available in hardware (typically via HPC) to detect malware. It has been argued that hardware malware schemes are desirable for two reasons: first, unlike software malware solutions that aim to protect vulnerable software with equally vulnerable software[1], hardware systems protect vulnerable software with robust hardware implementations that have lower bug defect density because of their simplicity. Second, while a motivated adversary can evade either defense, evasion is harder in a system that utilizes hardware features. The intuition is that the attacker does not have the same degree of control over lower-level hardware features as she has with software ones. For instance, it is easier to change system calls or file names than induce cache misses or branch misprediction in a precise way across a range of time scales while exploiting the system.

In this chapter, we introduce techniques to advance the use of lower-level microarchitectural features in the anomaly-based detection of malware exploits. Existing malware detection techniques can be classified along two dimensions: *detection approach* and the *malware features* they target, as presented in Figure 5.1. Detection approaches are traditionally categorized into misuse-based and anomaly-based detection. Misuse-based detection flags malware using preidentified attack signatures or heuristics. It can be highly accurate against known attacks but

[1]Software AV systems roughly have the same bug defect density as regular software.

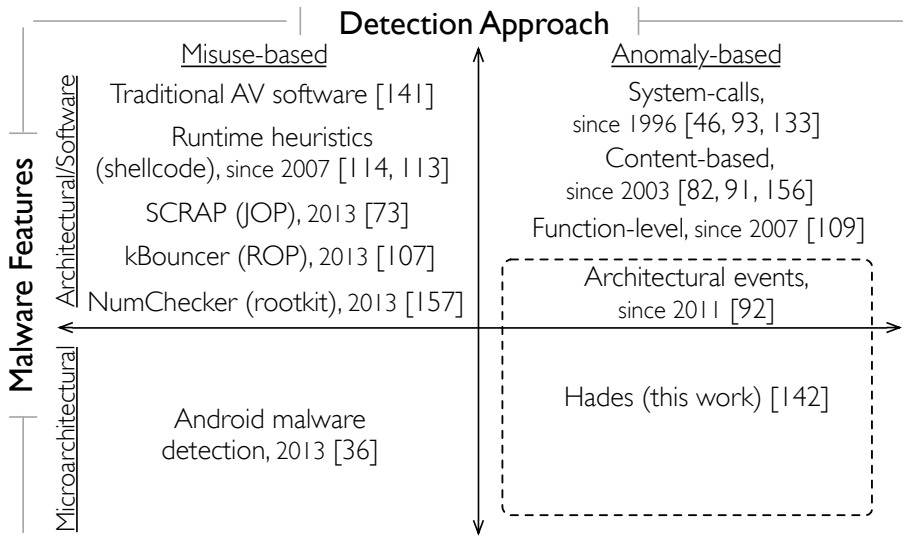

Figure 5.1: Taxonomy of malware detection approaches and some example works.

can be easily evaded with slight modifications that deviate from the signatures. On the other hand, anomaly-based detection characterizes baseline models of normalcy state and identifies attacks based on deviations from these models. Besides known attacks, it can potentially identify novel ones. There are a range of features that can be used for detection: until 2013, they were OS and application-level observables such as system calls and network traffic. Since then, lower-level features closer to hardware such as microarchitectural events have been used for malware detection. Shown in Figure 5.1, we examine for the first time, the feasibility and limits of anomaly-based malware detection using both architectural and low-level microarchitectural features available from HPCs.

Prior misuse-based research that uses microarchitectural features such as [37] focuses on flagging Android malicious apps by detecting payloads. A key distinction between our work and prior work is *when* the malware is detected. Malware infection typically comprises two stages, exploitation and take-over. In the exploitation stage, an adversary exercises a bug in the victim program to hijack control of the program execution. Exploitation is then followed by more elaborate take-over procedures to run a malicious payload such as a keylogger. Our work focuses on detecting malware during exploitation, as it not only gives more lead time for mitigations but

can also act as an early-threat detector to improve the accuracy of subsequent signature-based detection of payloads.

The key intuition for the anomaly-based detection of malware exploits stems from the observation that the malware, during exploitation, alters the original program flow to execute peculiar non-native code in the context of the victim program. Such unusual code execution tend to cause perturbations to the dynamic execution characteristics of the program. If these perturbations are observable, they can form the basis of detecting malware exploits.

In this chapter, we model the baseline characteristics of common vulnerable programs – Internet Explorer 8 and Adobe PDF Reader 9 (two of the most attacked programs) and examine if such perturbations do exist. Intuitively one might expect the deviations caused by exploits to be fairly small and unreliable, especially in vulnerable programs with extremely varied use such as in the ones we study. This intuition is validated in our measurements. On a Windows system using Intel x86 chips, our experiments indicate that distributions of measurements from the hardware performance counters are positively skewed, with many values being clustered near zero. This implies minute deviations caused by the exploit code cannot be effectively discerned directly. However, we show that this problem of identifying deviations from the heavily skewed distributions can be alleviated. We show that by using power transform to amplify small differences, together with temporal aggregation of multiple samples, we can identify the execution of the exploit within the context of the larger program execution. Further, in a series of experiments, we systematically evaluate the detection efficacy of the models over a range of operational factors, events selected for modeling and sampling granularity. For IE exploits, we can identify 100% of the exploitation epochs with 1.1% false positives. Since exploitation typically occurs across nearly 20 epochs, even with a slightly lower true positive rate, we can detect exploits with high probability. These are achieved at a sampling overhead of 1.5% slowdown using sampling rate of 512K instructions epochs.

Further we examine the resilience of our detection technique to evasion strategies of a more sophisticated adversary. We model *mimicry* attacks that craft malware to exhibit event char-

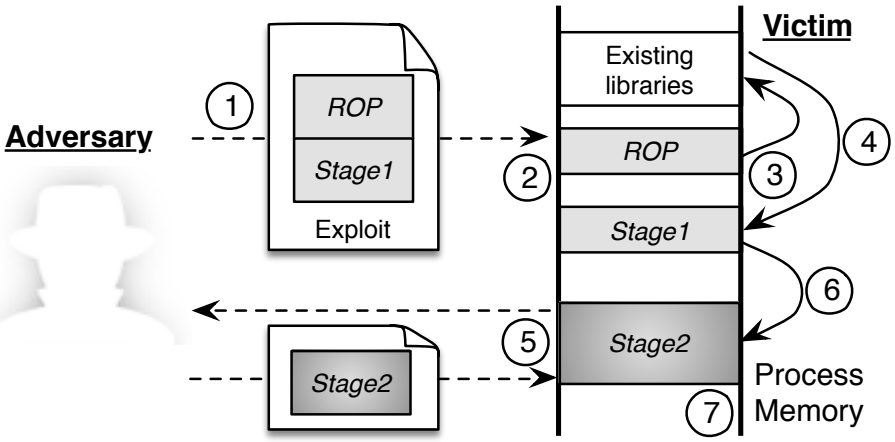

Figure 5.2: Multi-stage exploit process.

acteristics that resemble normal code execution to evade our anomaly detection models. With generously optimistic assumptions about attacker and system capabilities, we demonstrate that the models are susceptible to the mimicry attack. In a worst case scenario, the detection performance deteriorates by up to 6.5%. Due to this limitation we observe that anomaly detectors cannot be the only defensive solution but can be valuable as part of an ensemble of detectors that can include signature-based ones.

5.2 Background

Figure 5.2 shows a typical multi-stage malware infection process that results in a system compromise. The necessity for its multi-stage nature will become clear as we explain the exploit process in this section.

Triggering the vulnerability First the adversary crafts and delivers the exploit to the victim to target a specific vulnerability known to the adversary (Step ①). The vulnerability is in general a memory corruption bug; the exploit is typically sent to a victim from a webpage or a document attachment from an email. When the victim accesses the exploit, two exploit sub-programs, commonly known as the *ROP* and *Stage1* "shellcodes", load into the memory of the vulnerable program (Step ②). The exploit then uses the vulnerability to transfer control to the *ROP* shellcode

(Step ③).

Code Reuse Shellcode (*ROP*) To prevent untrusted data being executed as code, modern processors provide Data Execution Prevention (DEP) to restrict code from being run from data pages. To support JIT compilation however, DEP can be toggled by the program itself. So the *ROP*-stage shellcode typically circumvents DEP by reusing instructions in the original program binary – hence the name Code Reuse Shellcode – to craft a call to the function that disables DEP for the data page containing the next *Stage1* shellcode. The ROP shellCode then redirects execution to the next stage. (Step ④) [108].

***Stage1* Shellcode** This shellcode is typically a relatively small – from a few bytes to about 300 bytes[2] – code stub with exactly one purpose: to download a larger (evil) payload which can be run more freely. To maintain stealth, it downloads the payload in memory (Step ⑤).

***Stage2* Payload** The payload is the final piece of code that the adversary wants to execute on the target to perform a specific malicious task. The range of functionality of this payload, commonly a backdoor, keylogger, or reconnaissance program, is unlimited. After the payload is downloaded, the *Stage1* shellcode runs this payload as an executable using reflective DLL injection (Step ⑥), a stealthy library injection technique that does not require any physical files [45]. By this time, the victim system is fully compromised (Step ⑦).

The *Stage1* shellcode and *Stage2* payload are different in size, design and function, primarily due to the operational constraints on the *Stage1* shellcode. When delivering the initial shellcode in the exploit, exploit writers typically try to use as little memory as possible to ensure that the program does not unintentionally overwrite their exploit code in memory. To have a good probability for success, this code needs to be small, fast and portable, and thus is written in assembly language and uses very restrictive position-independent memory addressing style. These constraints limit the adversary's ability to write very large shellcodes. In contrast, the *Stage2* payload does not have all these constraints and can be developed like any regular program. This is similar to how OSes use small assembly routines to bootstrap and then switch to compiled code.

[2]As observed at `http://exploit-db.com`

The strategy and structure described above is representative of a large number of malware especially those created with recent web exploit kits [153]. These malware exploits execute completely from memory and in the process context of the host victim program. Further, they maintain disk and process stealth by ensuring no files are written to disk and no new processes are created, and thus easily evade most file based malware detection techniques.

5.3 Experimental Setup

Do the execution of different shellcode stages exhibit observable deviations from the baseline performance characteristics of the user programs? Can we use these deviations, if any, to detect a malware exploit as early as possible in the infection process? To address these questions, we conduct several feasibility experiments, by building baseline per-program models using machine learning classifiers and examining their detection efficacy over a range of operational factors. Here, we describe our experimental setup and detail how we collect and label the measurements attributed to different malware exploit stages.

5.3.1 Exploits

Unlike SPEC, there are no standard exploit benchmarks. We rely on a widely-used penetration testing tool *Metasploit* (from `www.metasploit.com`) to generate exploits for common vulnerable programs from publicly available information. We use exploits that target the security vulnerabilities *CVE-2012-4792*, *CVE-2012-1535* and *CVE-2010-2883* on IE 8 and the web plug-ins, *i.e.* Adobe Flash 11.3.300.257 and Adobe Reader 9.3.4 respectively. We choose to utilize *Metasploit* because the exploitation techniques it employs in the exploits are representative of multi-stage nature of real-world exploits.

Besides targeting different vulnerabilities using different ROP shellcode from relevant library files (`msvcrt.dll`, `icucnv36.dll`, `flash32.ocx`), we also vary both the *Stage1* (reverse_tcp, reverse_http, bind_tcp) shellcode and the *Stage2* final payload (meterpreter, vncinject, com-

mand_shell) used in the exploits.

Additionally, we instrument the start and end of the respective malware stages with debug trap *int3* instructions (`0xCC`) of one byte long, to label the exploit measurements with the respective stages solely for evaluation purposes.

5.3.2 Measurement Infrastructure

Since most real-world exploits run on Windows and PDF readers, and none of the architectural simulators can run programs of this scale, we use measurements from production machines. We develop a Windows driver to configure the performance monitoring unit on Intel i7 2.7GHz IvyBridge Processor to interrupt once every N instructions and collect the event counts from the HPCs. We also record the Process ID (PID) of the currently executing program so that we can filter the measurements based on processes.

We collect the measurements from a VMware Virtual Machine (VM) environment, installed with Windows XP SP3 and running a single-core with 512MB of memory. With the virtualized HPCs in the VM, this processor enables the counting of two fixed events (clock cycles, instruction retired) and up to a limit of four events simultaneously. We configure the HPCs to update the event counts only in the user mode. To ensure experiment fidelity for the initial study, measurements from the memory buffer are read and transferred via TCP network sockets to a recording program deployed in another VM. This recording program saves the stream of measurements in a local file that is used for our analysis.

5.3.3 Sampling Granularity

We experiment with various sampling interval of *N* instructions. We choose to begin the investigation with a sampling rate of every 512,000 instructions since it provides a reasonable amount of measurements without incurring too much overhead (See Section § 5.5.4 for an evaluation of the sampling overhead). Each sample consists of the event counts from one sampling time epoch, the identifying PID and the exploit stage label.

Since, unlike Linux, Windows does not provide a convenient mechanism to save and restore event counts across context switches, we note that our custom collection driver has the limitation of including event measurements from multiple processes in one sample. This lowers the fidelity of the measurements. To mitigate this measurement "contamination" issue, we rely on a rudimentary process-level filtering (by PID) and opting for progress-based sampling (*i.e.* sampling by retired instructions) over time-based sampling (*i.e.* by cycle count).

5.3.4 Labeling Measurements with Exploit Data

To investigate the effects of malicious code execution on the event characteristics, we require fine-grained labeling of the measurements captured during the various stages of an exploit. To this end, we instrumented the boundaries of the *ROP* and *Stage1* shellcode with debug trap *int3* instructions (*0xCC*) of one byte long, and installed an Interrupt Service Routine (ISR) to handle the debug traps. As the exploit code executes, the instrumented debug traps will be triggered and handled by the ISR updating an internal flag that is sampled together with the event counts. This setup allows us to identify accurately, with minimum perturbation to the exploit execution, the measurements attributed to various stages of the exploit code.

Instrumenting the boundaries of the ROP stage of an exploit is slightly trickier. We locate ROP gadgets of the instruction sequence "*int3 / ret*" and insert these gadgets at the start and end of the respective ROP shellcode.

With the above labels included in the measurements, we can then differentiate the measurements taken between different stages of an exploit. This insertion of the debug trap *int3* instructions is used solely for the labeling of the measurements for this study and is not used in a real-time detection setting. In Figure 5.3, we present a sample of the labeled event counts simultaneously recorded during an IE exploit. We label the event counts taken without the malware code exeuction as '*clean*'. Those event counts sampled while malware code is executing are labeled with the respective exploit stage names.

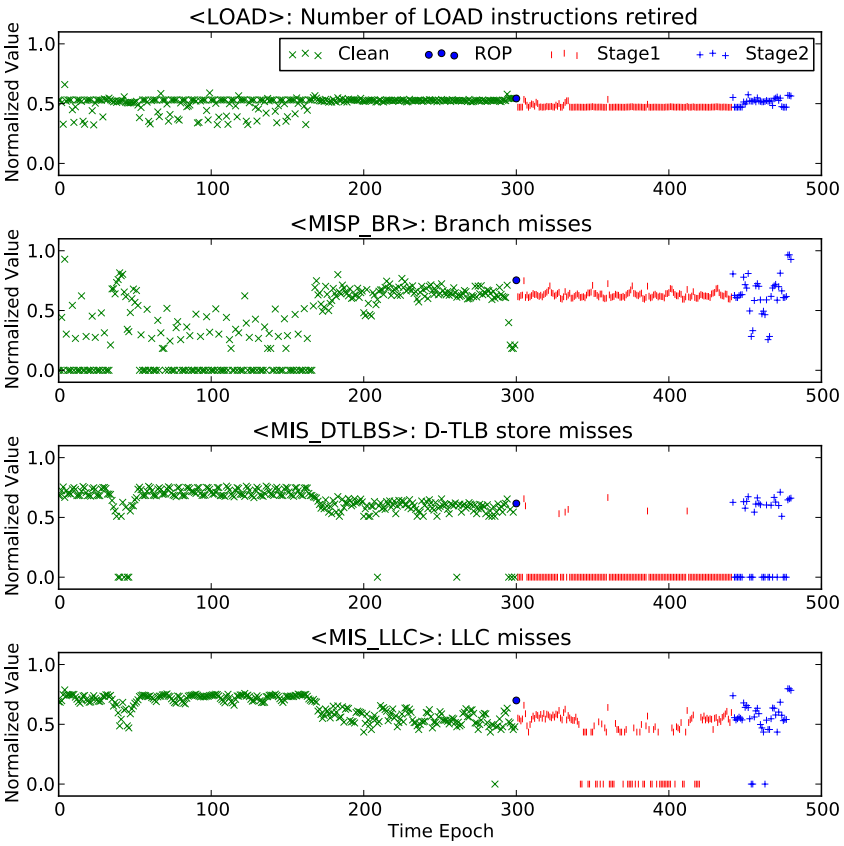

Figure 5.3: Labeled event counts (Sampled every 32k ins.)

5.3.5 Collection of Clean and Infected Measurements

To obtain clean exploit-free measurements for IE 8, we randomly browse websites that use different popular web plugins available on IE *viz.,* Flash, Java, PDF, Silverlight, and Windows Media Player extensions. We visit the top 20 websites from Alexa and include several other websites to widen the coverage of the use of the various plug-ins. Within the browser, we introduce variability by randomizing the order in which the websites are loaded across runs and by navigating the websites by clicking links randomly and manually on the webpages. The dynamic content on the websites also perturbs the browser caches. We use a maximum of two concurrent tabs. In addition, we simulate plug-in download and installation functions.

For Adobe PDF measurements, we download 800 random PDFs from the web, reserving half of them randomly for training and the other half for testing. To gather infected measurements,

we browse pages with our PDF exploits with the same IE browser that uses the PDF plug-in. We use Metasploit to generate these PDF exploits and ensure that both the clean and unclean PDFs have the same distribution of file types, for instance, same amount of Javascript.

We stop gathering infected measurements when we see creation of a new process. Usually the target process becomes unstable due to the corrupted memory state, and the malicious code typically "migrates" itself to another new or existing process to ensure persistence after the execution of the *Stage2* payload. This is an indication that the infection is complete.

While there are factors that may affect the results of our measurements, we take additional care to mitigate the following possible biases in our data during the measurement collection:

(1) **Between-run contamination**: After executing each exploit and collecting the measurements, we restore the VM to the state before the exploit is exercised. This ensures the measurements collected are independent across training and testing sets, and across different clean and exploit runs.

(2) **Exploitation bias**: Loading the exploits in the program in only one way may bias the sampled measurements. To reduce this bias, we collect the measurements while loading the exploit in different ways: (a) We launch the program and load the URL link of the generated exploit page. (b) With an already running program instance, we load the exploit page. (c) We save the exploit URL in a shortcut file and launch the link shortcut with the program.

(3) **Network condition bias**: The VM environment is connected to the Internet. To ensure that the different network latencies do not confound the measurements, we configure the VM environment to connect to an internally-configured *Squid* (from `www.squid-cache.org`) proxy and throttle the network bandwidth from 0.5 to 5Mbps using *Squid* delay pools. We vary the bandwidth limits while collecting measurements for both the exploit code execution and clean runs.

Table 5.1: Shortlisted candidate events to be monitored.

Architectural Events	
Name	**Event Description**
LOAD	Load instructions (ins.)
STORE	Store ins.
ARITH	Arithmetic ins.
BR	Branch (br.) ins.
CALL	All near call ins.
CALL_D	Direct near call ins.
CALL_ID	Indirect near call ins.
RET	Near return ins.

Microarchitectural Events	
Name	**Event Description**
LLC	Last level cache references
MIS_LLC	Last level cache misses
MISP_BR	Mispredicted br. ins.
MISP_RET	Mispred. near return ins.
MISP_CALL	Mispred. near call ins.
MISP_BR_C	Mispred. conditional br.
MIS_ICACHE	iCache misses
MIS_ITLB	iTLB misses
MIS_DTLBL	D-TLB load misses
MIS_DTLBS	D-TLB store misses
STLB_HIT	sTLB hits after iTLB misses
%MIS_LLC[a]	% of last level cache misses
%MISP_BR[a]	% of mispred. br.
%MISP_RET[a]	% of mispred. near RET ins.

[a]These *derived* events are not directly measured, but computed with two events measured by the HPCs. For example, %MISP_BR is computed as MISP_BR/BR.

5.4 Building Models

To use HPC measurements for anomaly-based detection of malware exploits, we need to build classification models to describe the baseline characteristics for each program we protect. These program characteristics are relatively rich in information and, given numerous programs, manually building the models is nearly impossible. Instead we rely on unsupervised machine learning techniques to dynamically learn possible hidden structure in these data. We then use this hidden structure – aka model – to detect deviations during exploitation.

We rely on a class of *unsupervised* one-class machine learning techniques for model building. The one-class approach is very useful because the classifier can be trained *solely* with measurements taken from a clean environment. This removes the need to gather measurements affected by exploit code, which is hard to implement and gather in practice. Specifically, we model the characteristics with the one-class Support Vector Machine (oc-SVM) classifier that uses the non-linear Radial Basis Function (RBF) kernel. In this study, the collection of the labeled measurements is purely for evaluating the effectiveness of the models in distinguishing the measurements taken in the presence of malware code execution.

5.4.1 Feature Selection

While the Intel processor we use for our measurements permits hundreds of events to be monitored using HPCs, not all of them are equally useful in characterizing the execution of programs. We examine most events investigated in previous program characterization works [132, 64], and various other events informed by our understanding of malware behavior. Out of the hundreds of possible events that can be monitored, we shortlist 19 events for this study in Table 5.1. We further differentiate between the *Architectural* events that give an indication of the execution mix of instructions in any running program, and the *Microarchitectural* ones that are dependent on the specific system hardware makeup.

Events with higher discriminative power The processor is limited to monitoring up to 4 events at any given time. Even with the smaller list of shortlisted events, we have to select only a subset of events, aka features, that can most effectively differentiate clean execution from infected execution. With the collected labeled measurements, we compute the Fisher Score (*F-Score*) to provide a quantitative measure of how effective a feature can discriminate measurements in clean executions from those in infected executions. The F-Score is a widely-used feature selection metric that measures the discriminative power of features [40]. A feature with better discriminative power would have a larger separation between the means and standard deviations for samples from different classes. The F-Score measures this degree of separation. The larger the F-Score, the more discriminative power the feature is likely to have. However, a limitation to using the F-Score is that it does not account for mutual information/dependence between features, but it can guide our selection of a subset of "more useful" features.

Since we are trying to differentiate samples with malicious code execution from those without, we compute the corresponding F-Scores for each event. Without loss of generality, we refer to the former as *negative* samples and the latter *positive* ones. For each feature *i*, let $n_{(+)}$ and $n_{(-)}$ denote the number of positive and negative test samples. μ, $\mu_{(+)}$ and $\mu_{(-)}$ are the averages of the i^{th} feature of all, positive and negative samples respectively. $\sigma^2_{(+)}$ and $\sigma^2_{(-)}$ denote the variances of the positive and negative samples respectively. The F-Score for the i^{th} feature, F_i is computed

as follows:

$$F_i = \frac{n_{(+)}(\mu_{(+)} - \mu)^2 + n_{(-)}(\mu_{(-)} - \mu)^2}{n_{(+)}\sigma^2_{(+)} + n_{(-)}\sigma^2_{(-)}}$$

where the numerator quantifies the between-class variance and the denominator the in-class variance for the i^{th} feature.

We compute the F-Scores for the different stages of malware code execution for each event and reduce the shortlisted events to the 7 top-ranked events for each of the two categories, as well as for the two categories combined, as shown in Table 5.2. Each row consists of the top-ranked events for an event category and the exploit stage.

We further select top 4 events from each row to form 9 candidate event sets that we will use to build the baseline characteristic models of the IE browser. Each model constructed with one set of events can then be evaluated for its effectiveness in the detection of various stages of malware code execution. For brevity, we assign a label (such as *A-0* and *AM-2*) to each set of 4 events in Table 5.2 and refer to each model based on this *set label*. We note that the derived events such as %MISP_BR are listed in the table solely for comparison. Computing them requires monitoring two events and reduces the number of features used in the models. Via experimentation, we find that using them in the models does not increase the efficacy of the models. Thus, we exclude them from the event sets.

Feature Extraction Each sample consists of simultaneous measurements of all the four event counts in one time epoch. We convert the measurements in each sample to the vector subspace, so that each classification vector is represented as a four-feature vector. Each vector, using this feature extraction method, represents the measurements taken at the smallest time-slice for that sampling granularity. These features will be used to build *non-temporal* models.

Since we observe that malware shellcode typically runs over several time epochs, there may exist temporal relationships in the measurements that can be exploited. To model any potential temporal information, we extend the dimensionality of each sample vector by grouping the *N* consecutive samples and combining the measurements of each event to form a vector with $4N$ features. We use $N = 4$ to create sample vectors consisting of 16 features each, so each sample

Table 5.2: Top 7 most discriminative events for different stages of exploit execution (Each event set consists of 4 event names in **Bold**. E.g, monitoring event set *A-0* consists of simultaneously monitoring Ret, Call_D, Store and Arith event counts.)

Exploit Stage	Set Label	Events ranked by F-scores						
		1	2	3	4	5	6	7
Architectural Events								
ROP	A-0	**Ret**	**Call_D**	**Store**	**Arith**	Call	Load	Call_Id
Stage1	A-1	**Store**	**Load**	**Call_ID**	**Ret**	Call_D	Call	Arith
Stage2	A-2	**Store**	**Call_ID**	**Ret**	**Call_D**	Call	Arith	Br
Microarchitectural Events								
ROP	M-0	**Misp_Br_C**	%Misp_Br	**Misp_Br**	%Misp_Ret	**Mis_Itlb**	**Mis_Llc**	Mis_Dtlbs
Stage1	M-1	**Misp_Ret**	**Misp_Br_C**	%Misp_Ret	%Misp_Br	**Mis_Dtlbs**	**Stlb_Hit**	Misp_Br
Stage2	M-2	**Misp_Ret**	**Stlb_Hit**	**Mis_Icache**	**Mis_Itlb**	%Misp_Ret	Misp_Call	Mis_Llc
Both Architectural and Microarchitectural Events								
ROP	AM-0	**Misp_Br_C**	%Misp_Br	**Misp_Br**	%Misp_Ret	**Mis_Itlb**	**Ret**	Mis_Llc
Stage1	AM-1	**Store**	**Load**	**Misp_Ret**	**Call_ID**	Ret	Call_D	Call
Stage2	AM-2	**Store**	**Call_ID**	**Misp_Ret**	**Ret**	Call_D	Call	Stlb_Hit

vector effectively represents measurements across 4 time epochs. By grouping samples across several time epochs, we use the synthesis of these event measurements to build *temporal* models.

With the granularity at which we sample the measurements, the execution of the ROP shellcode occurs within the span of just one sample. Since we are creating vectors with a number of samples as a group, the ROP payload will only contribute to one small portion of a vector sample. So we leave out the ROP shellcode for testing using this form of feature extraction.

5.5 Results

5.5.1 Anomalies Not Directly Detectable

We first investigate if we can gain insights into the distribution of the event counts for a clean environment and one attacked by an exploit. Without assuming any prior knowledge of the distributions, we use box-and-whisker[3] plots of normalized measurements for different events. These plots offer a visual gauge of the range and variance in the measurements and an initial indi-

[3]The box-and-whisker plot is constructed with the bottom and top of the box representing the first and third quartiles respectively. The red line in the box is the median. The whiskers extend to 1.5 times the length of the box. Any outliers beyond the whiskers are plotted as blue + ticks.

cation on how distinguishable the measurements taken with the execution of different malware code stages are from the *clean* measurements from an exploit-free environment.

These distribution comparisons suggest that any event anomalies manifested by malware code execution are not trivially detectable, due to two key observations. (1) Most of the measurement distributions are very positively skewed, with many values clustered near zero. (2) Deviations, if any, from the baseline event characteristics due to the exploit code are not easily discerned.

5.5.2 Power Transform

To address this challenge, we rely on rank-preserving power transform on the measurements to positively scale the values. In the field of statistics, the power transform is a common data analysis tool to transform non-normally distributed data to one that can be approximated by a normal distribution. Used in our context, it has the value of magnifying any slight deviations that the malware code execution may have on the baseline characteristics.

For each event type, we find the appropriate power parameter λ such that the normalized median is roughly 0.5. For each event *i*, we maintain and use its associated parameter λ_i to scale all its corresponding measurements throughout the experiment. Each normalized and scaled event measurement for event *i*, normalized_i, is transformed from the raw value (raw_i), minimum value (min_i), maximum value (max_i) as follows:

$$\text{normalized}_i = \left(\frac{\text{raw}_i - \text{min}_i}{\text{max}_i}\right)^{\lambda_i} \tag{5.1}$$

Using this power transform, we plot the distributions of all the events, in Figure 5.4. Now we observe varying deviations from baseline characteristics due to different stages of malware code execution for various event types. Some events (such as MISP_RET and STORE) show relatively larger deviations, especially for the *Stage1* exploit shellcode. These events likely possess greater discriminative power in indicating the presence of malware code execution. Clearly, there are also certain events that are visually correlated. The RET and CALL exhibit similar distributions.

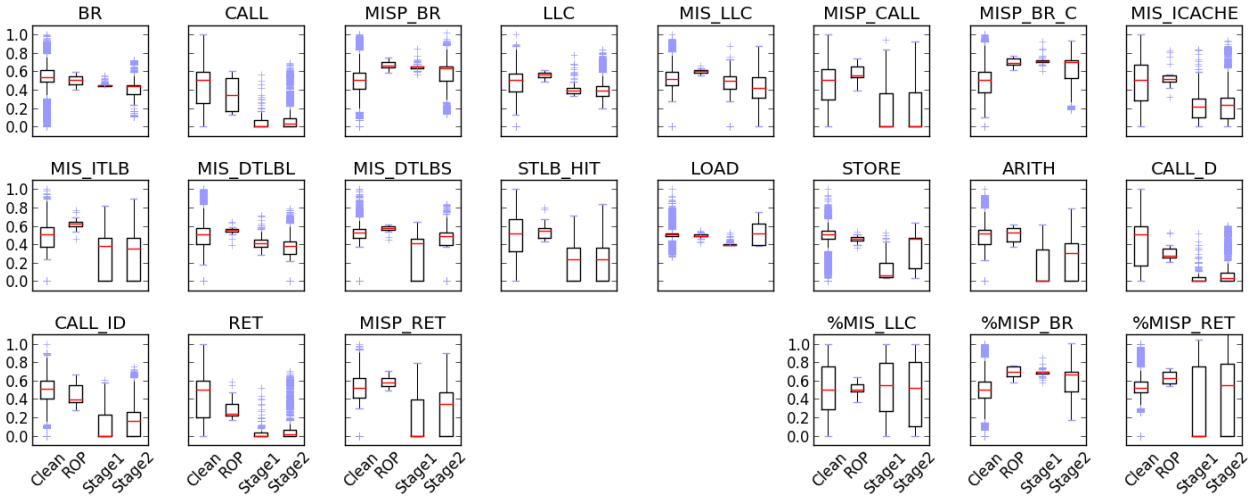

Figure 5.4: Distribution of events (*after* power transform) with more discernible deviations.

We can also observe strong correlation between those computed events (such as %Misp_Br) and their constituent events (such as Misp_Br).

5.5.3 Evaluation Metrics for Models

To visualize the classification performance of the models, we construct the *Receiver Operating Characteristic* (ROC) curves which plot the percentage of truely identified malicious samples (True positive rate) against the percentage of *clean* samples falsely classified as malicious (False positive rate). Each sample in the non-temporal model corresponds to the set of performance counter measurements in one epoch; each temporal sample spans over 4 epochs. Furthermore, to contrast the relative performance between the models in the detection of malicious samples, the area under the *ROC* curve for each model can be computed and compared. This area, commonly termed as the *Area Under Curve* (AUC) score, provides a quantitative measure of how well a model can distinguish between the clean and malicious samples for varying thresholds. The higher the AUC score, the better the detection performance of the model.

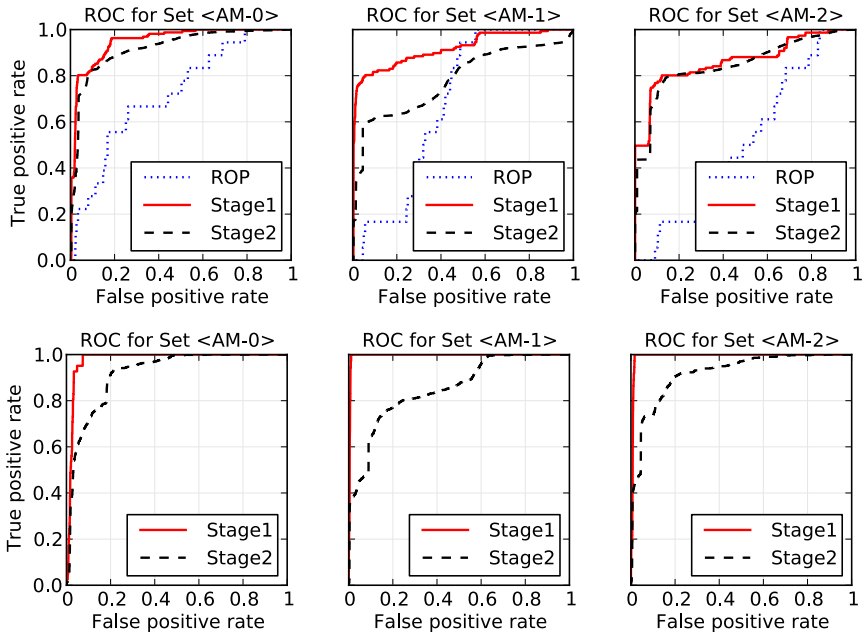

Figure 5.5: **Top**: ROC plots for *Non-Temporal* 4-feature models for IE. **Bottom**: ROC plots for *Temporal* 16-feature models for IE.

5.5.4 Detection Performance of Models

We first build the oc-SVM models with the training data, and evaluate them with the testing data using the non-temporal and temporal modeling on the nine event sets. To characterize and visualize the detection rates in terms of true and false positives over varying thresholds, we present the ROC curves of both approaches in Figure 5.5. For brevity, we only present the ROC curves for models that use both architectural and microarchitectural events. We also present the overall detection results in terms of AUC scores in Figure 5.6 and highlight the key observations that affect the detection accuracy of the models below.

Different Stages of Malware Exploits We observe that the models, in general, perform best in the detection of the *Stage1* shellcode. These results suggest the *Stage1* shellcode exhibits the largest deviations from the baseline architectural and microarchitectural characteristics of benign code. We achieve a best-case detection accuracy of 99.5% for *Stage1* shellcode with *AM-1* models.

On the other hand, the models show mediocre detection capabilities for the ROP shellcode. The models does not perform well in the detection of the ROP shellcode, likely because the sam-

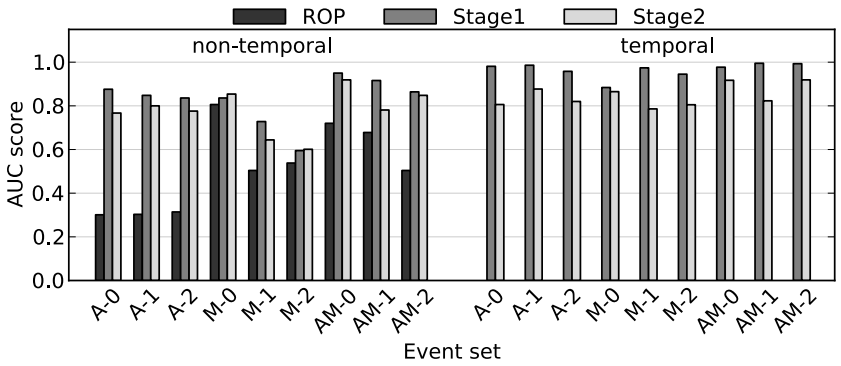

Figure 5.6: Detection AUC scores for different event sets using non-temporal and temporal models for IE.

pling granularity at 512k instructions is too coarse-grained to capture the deviations from the ROP shellcode in the baseline models. While the *Stage1* and *Stage2* shellcode executes within several time epochs, we measured that the ROP shellcode takes 2182 instructions on average to complete execution. It ranges from as few as 134 instructions (for the Flash ROP exploit) to 6016 instructions (for the PDF ROP exploit). Since we are keeping the sampling granularity constant, the sample that contains measurements during the ROP shellcode execution will largely consist of samples from the normal code execution.

Non-Temporal vs Temporal Modeling We observe that the detection accuracy of the models for all event sets improves with the use of temporal information. By including more temporal information in each sample vector, we reap the benefit of magnifying any deviations that are already observable in the non-temporal approach. For event set *M-2*, this temporal approach of building the models improves the AUC score from the non-temporal one by up to 58.8%.

Architectural vs Microarchitectural Events We quantify the detection capabilities of our models by considering the architectural and microarchitectural features separately and in combination. Models built using only architectural events achieve AUC scores on average 4.1% better than those built solely with microarchitectural events. Combining the use of microarchitectural events with architectural ones improves the average AUC scores by 5.8% and 1.4% for microarchitectural-only and architectural-only models respectively. It is more advantageous to incorporate the use of both types of events in the detection models. For instance, by selecting

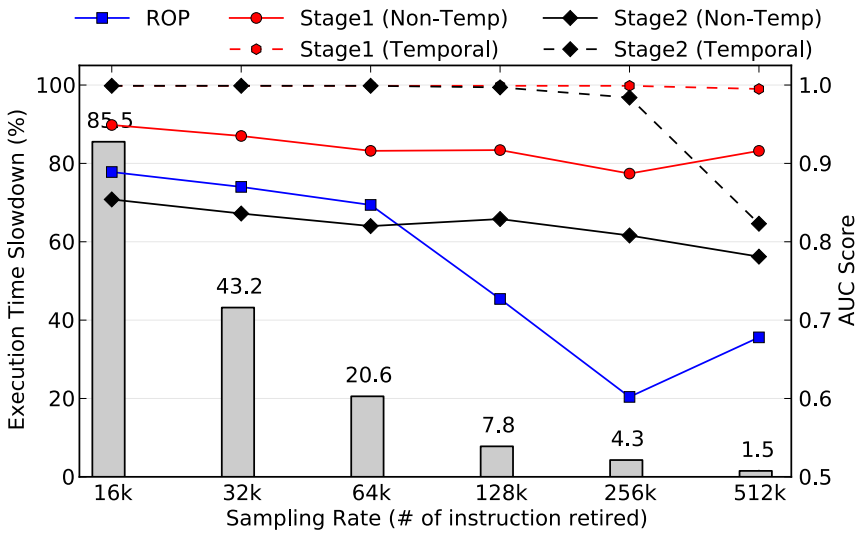

Figure 5.7: Trade-off between sampling overhead for different sampling rates versus detection accuracy using set *AM-1*.

and modeling both the most discriminative architectural and microarchitectural events together, we can achieve higher detection rates of up to an AUC score of 99.5% for event set *AM-1*.

Different Sampling Granularities While we use the sampling rate of 512K instructions for the above experiments, we also examine the impact on detection efficacy for various sampling granularities. Although the hardware-based HPCs incur a near-zero overhead in the monitoring of the event counts, a pure software-only implementation of the detector still requires running programs to be interrupted periodically to sample the event counts. This inadvertently leads to a slowdown of the overall running time of programs due to this sampling overhead. To inform the deployment of a software-only implementation of such a detection paradigm, we evaluate the sampling performance overhead for different sampling rates.

To measure this overhead, we vary the sampling granularity and measure the slowdown in the programs from the SPEC 2006 benchmark suite. We also repeat the experiments using the event set *AM-1* to study the effect of sampling granularity has on the detection accuracy of the model. We plot the execution time slowdown over different sampling rates with the corresponding detection AUC scores for various malware exploit stages in Figure 5.7.

We observe that the detection performance generally deteriorates with coarser-grained sam-

pling. This illustrates a key limitation of the imprecise sampling technique used on Windows systems. For example, during the span of instructions retired in one sample, while we may label these measurements as belonging to a specific process PID, these measurements may also contain measurements belonging to other processes context-switched in and out during the span of this sample. The interleaved execution of different processes creates this "noise" effect that becomes more pronounced with a coarser-grained sampling rate and deteriorates the detection performance. Nonetheless, we note that the reduction in sampling overhead at coarser-grained rates far proportionately outstrips the decrease in detection performance.

Constrained Environments To further investigate the impact of the aforementioned "noise" effect, we also assess the impact on detection accuracy in the scenario where we deploy both the online classification and the measurement gathering in the same VM. As described in Section § 5.3.2, we collect the measurements in our study from one VM and transfer the measurements to the recorder in another VM to be saved and processed. We term this cross-remote-VM scenario where the sampling and the online classification are performed on different VMs as *R-1core*.

For this experiment, we use the event model set *AM-1* using two additional local-VM scenarios utilizing both one and two cores separately. We term these two scenarios as *L-1core* and *L-2core* respectively. We present the detection AUC scores for the three different scenarios in Table 5.3 (*Left*).

We observe the detection performance suffers when the online classifier is deployed locally together with the sampling driver. This may be due to possible noise introduced to the event counts while the online detector is executing and processing the stream of samples. This highlights a key limitation of the current method of periodic collection of HPC measurements on Windows systems, where we are unable to cleanly segregate the measurements on a per-process basis.

To alleviate this problem, we envision a software-only implementation on a distributed or multi-core system in which the online detector is running separately from the system or core being protected. Furthermore, since this detection approach requires little more than a stream

Table 5.3: AUC scores for: **(Left)** Constrained scenarios for IE using set *AM-1* and **(Right)** Standalone Adobe PDF Reader.

Scenario	Non-Temporal			Temporal	
Label	ROP	Stage1	Stage2	Stage1	Stage2
L-1core	0.505	0.895	0.814	0.918	0.900
L-2core	0.496	0.890	0.807	0.907	0.813
R-1core	0.678	0.916	0.781	0.995	0.823

Set	Non-Temporal			Temporal	
Label	ROP	Stage1	Stage2	Stage1	Stage2
AM-0	0.931	0.861	0.504	0.967	0.766
AM-1	0.857	0.932	0.786	0.999	0.863
AM-2	0.907	0.939	0.756	0.998	0.912

of HPC measurements, this makes it suitable as an out-of-VM deployment in a Virtual Machine Introspection (VMI)-based setting [50] for intrusion detection. This approach requires minimum guest data structures, relieving the need to bridge the semantic gap, a common problem faced by VMI works. Another potential avenue to alleviate the "noise" problem is a pure hardware implementation using a separate and secure dedicated core or co-processor for the execution of an online detector as proposed in [37].

5.5.5 Results for Adobe PDF Reader

Due to space constraints, we do not present the full results from our experiments on the standalone Adobe PDF Reader. We present the AUC detection performance of the models built with the event sets *AM-0,1,2* in Table 5.3 (*Right*). Compared to the models for IE, the detection of ROP and *Stage1* shellcode generally improves for the Adobe PDF Reader. We even achieve an AUC score of 0.999 with the *AM-1* temporal model. The improved performance of this detection technique for the PDF Reader suggests that its baseline characteristics are more stable given the less varied range of inputs it handles compared to IE.

5.6 Analysis of Evasion Strategies

In general, anomaly-based intrusion detection approaches, such as ours, are susceptible to *mimicry* attacks. To evade detection, a sophisticated adversary with sufficient information about the anomaly detection models can modify her malware into an equivalent form that exhibits similar baseline architectural and microarchitectural characteristics as the normal programs. In this

section, we examine the degree of freedom an adversary has in crafting a mimicry attack and how it impacts the detection efficacy of our models.

Adversary Assumptions We assume the adversary (a) knows all about the target program such as the version and OS to be run on, and (b) is able to gather similar HPC measurements for the targeted program to approximate its baseline characteristics. (c) She also knows the way the events are modeled, but *not* the exact events used. We highlight three ways the adversary can change her attack while retaining the original attack semantics.

Assumption (c) is realistic, given the hundreds of possible events that can be monitored on a modern processor. While she may uncover the manner the events are modeled, it is difficult to pinpoint the exact subset of four events used given the numerous possible combinations of subsets. Furthermore, even if the entire event list that can be monitored is available, there may still exist some events (such as events monitored by the power management units) that are not publicly available. Nonetheless, to describe attacks 1 and 2, we optimistically assume the adversary has full knowledge of all the events that are used in the models.

Attack 1: Padding The first approach is to pad the original shellcode code sequences with "no-op" (no effect) instructions with a sufficient number so that the events manifested by the shellcode match that of the baseline execution of the program. These no-op instructions should modify the measurements for all the events monitored, in tandem, to a range acceptable to the models.

The adversary needs to know the events used by the model *a priori*, in order to exert an influence over the relevant events. We first explore feasibility of such a mimicry approach by analyzing the *Stage1* shellcode under the detection model of event set *AM-1*. After studying the true positive samples, we observe that the event characteristics exhibited by the shellcode are due to the unusually low counts of the four events modeled. As we re-craft the shellcode at the assembly code level to achieve the mimicry effect, we note three difficulties.

(1) **Multi-instruction no-ops**: Some microarchitectural events require more than one instruction to effect a change. For example, to raise the MISP_RET counts, sequences of RET code need

to be crafted in a specific order. Insertion of no-ops must be added in multi-instruction segments.

(2) **Event co-dependence**: To maintain the original shellcode semantics, certain registers need to be saved and subsequently restored. These operations constitute STORE /LOAD μ-operations and can inadvertently affect both STORE and LOAD events. Thus we are rarely able to craft no-op code segments to modify each event independently. For instance, among the events in *AM-1*, only the no-op instruction segment for STORE can be crafted to affect it independently. Event co-dependence makes adversarial control of values of individual events challenging.

(3) **No-op insertion position**: Insertion position of the no-op instruction segments can be critical to achieve the desired mimicry effect. We notice the use of several loops within the shellcode. If even one no-op segment is inserted into the loops, that results in a huge artificial increase in certain event types, consequently making that code execution look more malicious than usual.

Next, we examine the impact of such mimicry efforts on the detection performance. We pad the *Stage1* shellcode at random positions (avoiding the loops) with increasing number of each crafted no-op instruction segment and repeated the detection experiments. In Figure 5.8 (*Left*), we plot the box-and-whisker plots of the anomaly scores observed from the samples with varying numbers of injected no-op code. In general, the anomaly scores become less anomalous with the padding, until after a tipping point where inserting too many no-ops reverses mimicry effect. In the same vein, we observe in Figure 5.8 (*Right*) that the detection AUC scores decrease as the samples appear more normal. For the worst case, the detection performance suffers by up to 6.5% just by inserting *only* the CALL_ID no-ops. We do not study combining the no-ops for different events, but we believe it should deteriorate the detection performance further.

Attack 2: Substitution Instead of padding no-ops into original attack code sequences, the adversary can replace her code sequences with equivalent variants using code obfuscation techniques, common in metamorphic malware [25]. Like the former attack, this also requires that she knows the events used by the models *a priori*. To conduct this attack, she must first craft or generate equivalent code variants of code sequences in her exploits, and profile the event char-

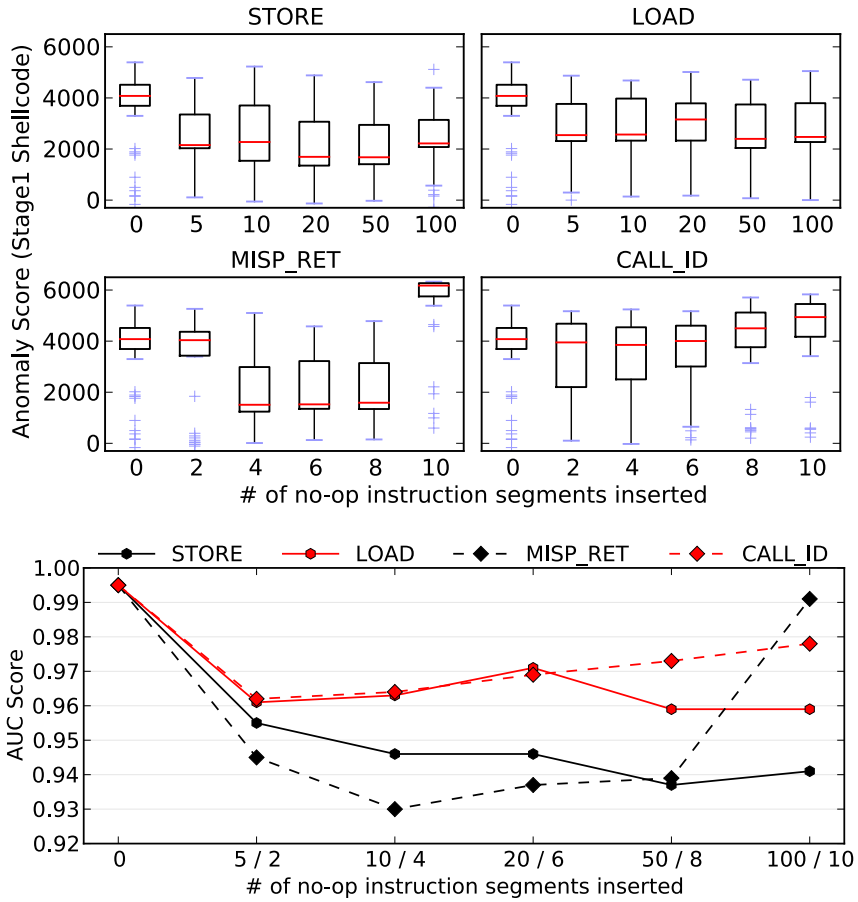

Figure 5.8: Impact of inserting no-op segments on: **(Left)** The anomaly scores of *Stage1* shellcode and **(Right)** The detection efficacy of *Stage1* shellcode.

acteristics of each variant. She can adopt a greedy strategy by iteratively substituting parts of her attack code with the equivalent variants, measuring the HPC events of the shellcode and ditching those variants that exhibit characteristics not acceptable to the models. However, while this greedy approach will terminate, it warrants further examination as to whether the resulting shellcode modifications suffice to evade the models. We argue that this kind of shellcode re-design is hard and will substantially raise the bar for exploit writers.

Attack 3: Grafting This attack requires either inserting benign code from the target program directly into the exploit code, or co-scheduling the exploit shellcode by calling benign functions (with no-op effects) within the exploit code. This attack somewhat *grafts* its malicious code execution with the benign ones within the target program, thus relieving the need for the knowledge

of the events that are modeled. If done correctly, it can exhibit very similar characteristics as the benign code it grafts itself to. As such this represents the most powerful attack against our detection approach.

While we acknowledge that we have not crafted this form of attack in our study, we believe that it is extremely challenging to craft such a grafting attack due to the operational constraints on the exploit and shellcode, described in Section § 5.2. (1) Inserting sufficient benign code into the shellcode may exceed the vulnerability-specific size limits and cause the exploit to fail. (2) To use benign functions for the grafting attacks, these functions have to be carefully identified and inserted so that they execute sufficiently to mimic the normal program behavior and yet not interfere with the execution of the original shellcode. (3) The grafted code must not unduly increase the execution time of the entire exploit.

5.6.1 Defenses

Unlike past anomaly-based detection systems that detect deviations based on the syntactic/semantic structure and code behavior of the malware shellcode, our approach focuses on the architectural and microarchitectural side-effects manifested through the code execution of the malware shellcode. While the adversary has complete freedom in crafting her attack instruction sequences to evade the former systems, she cannot directly modify the events exhibited by her attack code to evade our detection approach. To conduct a mimicry attack here, she has to carefully "massage" her attack code to manifest a combination of event behaviors that are accepted as benign/normal under our models. This second-order degree of control over the event characteristics of the shellcode adds difficulty to the adversary's evasion efforts. On top of this, we discuss further potential defense strategies to mitigate the impact of the mimicry attacks.

Randomization Introducing secret randomizations into the models has been used to strengthen robustness against mimicry attacks in anomaly-based detection systems [159]. In our context, we can randomize the events used in the models by training multiple models using different subsets of the shortlisted events. We can also randomize the choice of model to utilize

over time. Another degree of randomization is to change the number of consecutive time-epoch samples to use for each sample for the temporal models. In this manner, the adversary does not know which model is used during the execution of her attack shellcode. For her exploit to be portable and functional on a wide range of targets, she has to modify her shellcode using the no-op padding and instruction substitution mimicry attacks for a wider range of events (and not just the current four events).

Multiplexing At the cost of higher sampling overhead, we can choose to sample at a finer sampling granularity and measure more events (instead of the current four) by multiplexing the monitoring – we can approximate the simultaneous monitoring of 8 events across two time epochs by monitoring 4 events in one and another 4 in the other. This increases to the input dimensionality used in the models, making it harder for the adversary to make all the increased number of monitored event measurements appear non-anomalous.

Defense-in-depth Consider a defense-in-depth approach, where this malware anomaly detector using HPC manifestations is deployed with existing anomaly-based detectors monitoring for other features of the malware, such as its syntactic and semantic structure [159, 84, 94] and its execution behavior at system-call level [135, 47, 96, 128] and function level [110]. In such a setting, in order for a successful attack, an adversary is then forced to shape her attack code to conform to normalcy for each anomaly detection model. An open area of research remains in quantifying this multiplicative level of security afforded by the combined use of these HPC models with existing defenses, *i.e.* examining the difficulty in shaping the malware shellcode to evade detectors using statistical and behavioral software features, while simultaneously not exhibiting any anomalous HPC event characteristics during execution.

5.7 Discussion

Hades relies on readily available hardware performance counters to profile microarchitectural behavior of programs. Researchers have highlighted issues (non-determinism, inter-run vari-

ations and overcount among all) in the use of performance counters to provide accurate and deterministic measurements [164, 165]. In this section, we discuss such characteristics of hardware performance counters and how they impact the collection of microarchitectural events for malware detection.

Variability in Measurements While hardware performance counters should ideally provide exact and deterministic (*i.e.* getting consistent results across different runs), real-world implementations fall short of this ideal expectation. Weaver *et al.* show that events collected with the performance counters exhibit run-to-run variation and persistent overcount, even when conducted in strictly controlled environments [165]. They note that these variations are influenced by non-deterministic hardware interrupts and page faults. These sources of variations can be hard to predict, since they depend on per-machine per-run system conditions like the OS activity, instruction-specific quirks, processor errata or I/O interference. Furthermore, such variations can exist both within the same machine or across multiple machines [164].

Mitigating Noisy Measurements The most direct impact of measurement variations is the introduction of noise into the malware detection models. The higher the degree of noise, the less stable the models and the greater the false positive rate in detection performance. Unlike traditional uses (deterministic replay, architectural simulation) of performance counters that require deterministic measurements, our use of machine learning models in profiling programs accommodate some degree of noise – the models remain useful as long as the perturbations caused by malicious code execution exceed in magnitude that of the noise, as our experiments have shown. Our feature selection process using F-Score (§ 5.4.1) allows us to shortlist events with the best signal-to-noise ratio and with the best chance of overcoming the aforementioned problem of *noise*. The intra-run variability can also be attributed to the imprecise sampling of events, more commonly known as *sample skid* [163]. The problem of *skid* can be partially mitigated with performance counter functionalities that precisely attribute event counts across interrupts. Examples of such hardware features are Intel Precise Event-Based Sampling (PEBS) and AMD Instruction Based Sampling (IBS) [138]. Furthermore, systematic inaccuracies caused by event

overcount are accounted for by training and using malware detection models on a per-machine basis; if events are consistently under- or over-estimated on a machine, they will have negligible impact on the anomaly detection models.

Inter-machine Variations Weaver *et al.* report inter-machine variations for the instruction counts event [164]. In our per-machine training and detection deployment, such variations can work in our advantage, towards making it harder for attacker to game the detection models. Assuming such inter-machine variations exist, each per-machine trained model will be very unique to each system makeup. As a result, creating *ex-ante* mimicry attacks for machine-specific models becomes very challenging; the attacker will be forced to make assumptions on the machine-specific hardware characteristics that drive the underlying anomaly detection models.

5.8 Architectural Enhancements for Malware Detection

Performance counters are typically used for low-level performance analysis and tuning, and for program characterization. In this section, we suggest some simple modifications to extend their benefits for detecting malware based on anomalies.

More performance counters Our experiments show that adding events can help better distinguish between benign and malicious code execution. Expanding the set of performance counters that can be monitored concurrently can potentially increase detection fidelity. Cheap hardware mechanisms to observe instruction and data working set changes, and basic-block level execution frequencies can improve malware detection accuracies further.

Interrupt-less periodic access Currently reading performance counters requires the host process to be interrupted. This leads in expensive interrupt-handling cost and undue sampling overhead to the programs. If the performance monitoring units are re-designed with the capability to store performance counter measurements periodically to a designated memory region without generating interrupts, accessing the samples from this region directly will eliminate the sampling overhead. Most importantly, this allows for monitoring at finer granularities to reduce

the ”noise” effect described in § 5.5.4, and leaves greater scope for better detection.

Custom Accelerators In our work we sample at a very coarse granularity of 512K instructions. Results show that finer granularity sampling can improve detection accuracies. Currently the detector is implemented in software, but at much finer granularities, to keep up with increased data volumes, hardware implementations will likely be necessary and certainly be more energy-efficient compared to software implementations.

Secret Events In this work we have used publicly available performance counters for detecting malware. The malware detector can be built just as well with non-public microarchitectural events. Keeping the events secret increases the difficulty of the attacker to conduct evasion attacks. This model is very similar to how on-chip power controllers operate in modern processors. In the latest Intel and AMD processors, an on-chip microcontroller receives activity factors from various blocks on the chip and uses this information to make power management decisions. Neither the units providing activity factors or the logic/algorithm for making power management decisions are public information, and has been hard to reverse engineer. Further the power management algorithm is not directly accessible to software but during emergencies an exception is delivered to the software. A similar model can be used to build malware detectors.

5.9 Related Work

The use of low-level hardware features for malware detection (instead of software ones) is a recent development. Demme *et al.* demonstrate the feasibility of misuse-based detection of Android malware programs using microarchitectural features [37]. While they model microarchitectural signatures of malware programs, we build baseline microarchitectural models of benign programs we are protecting and detect deviations caused by a potentially wider range of malware (even ones that are previously unobserved). Another key distinction is that we are detecting malware shellcode execution of an exploit within the context of the victim program during the act of exploitation; they target Android malware as whole programs. After infiltrating the system via

an exploit, the malware can be made stealthier by installing into peripherals, or by infecting other benign programs. Stewin *et al.* propose detecting the former by flagging additional memory bus accesses made by the malware [140]. Malone *et al.* examine detecting the latter form of malicious static and dynamic program modification by modeling the architectural characteristics of benign programs (and excluding the use of microarchitectural events) using linear regression models [95]. Another line of research shows that malware can be detected using side-channel power perturbations they induce in medical embedded devices [31], software-defined radios [52] and mobile phones [78]. However, Hoffman *et al.* show that the use of such power consumption models can be very susceptible to noise, especially in a device with such widely varied use as the modern smartphone [61].

Besides HPCs, several works have leveraged other hardware facilities on modern processors to monitor branch addresses efficiently to thwart classes of exploitation techniques. kBouncer uses the Last Branch Recording (LBR) facility to monitor for runtime behavior of indirect branch instructions during the invocation of Windows API for the prevention of ROP exploits [108]. To enforce control flow integrity, CFIMon [174] and Eunomia [176] leverage the Branch Trace Store (BTS) to obtain branch source and target addresses to check for unseen pairs from a pre-identified database of legitimate branch pairs. Unlike our approach to detecting malware, these works are designed to prevent exploitation in the first place, and are orthogonal to our anomaly detection approach.

5.10 Conclusions

This work introduces the novel use of hardware-supported lower-level microarchitectural features to the anomaly-based detection of malware exploits. This represents the first work to examine the feasibility and limits of using unsupervised learning on microarchitectural features from HPCs to detect malware. We demonstrate that the dynamic execution of commonly attacked programs can be efficiently characterized with minimal features – the stream of event measurements

easily accessible from the HPC, and used to detect lower-level perturbations caused by malware exploits to the baseline characteristics of benign programs. Unlike its misuse-based counterparts previously proposed, this anomaly-based detection approach can detect a wider range of malware, even novel ones. This work can thus be used in concert with its misuse-based counterparts to better security. Further, in modeling a class of potential mimicry attacks against our detector, we show that it can be challenging for an adversary to precisely control these hardware features to conduct an evasion attack.

6

Conclusion

This dissertation puts forth the thesis that full-system security can be improved by examining and leveraging the interworking of hardware and software. First, I motivate security-aware energy management designs by highlighting multiple issues in current designs of energy management mechanisms (Chapter 3: CLKscrew). I show that unfettered software access to isolation-agnostic hardware regulators exposes security-sensitive trusted execution environments to practical and realizable security risks. Second, I design and implement the Destructive Code Read primitive as a novel defense to destroy runtime information crucial for exploitation (Chapter 4: Heisenbyte). Leveraging hardware virtualization support, the primitive extends the benefits of execute-only memory defenses to COTS systems. Last but not least, in the third case study, I describe a hardware performance counter-based framework that efficiently profiles previously untapped microarchitectural interaction between hardware and software to detect anomalous and malicious code execution (Chapter 5: Hades). These case studies underscore the importance of giving commodity hardware-software interfaces due consideration when designing robust defenses or architecting systems for security.

6.1 Future Directions

As systems become more heterogeneous (*e.g.* mobile SoCs), widespread (*e.g.* IoT) and multi-layered (*e.g.* disparate privilege modes), ensuring security across the computing stack becomes

increasingly crucial. To this end, I introduce novel techniques to leverage readily available hardware features exposed to software to secure systems. These techniques use commodity features, making them deployable on systems in the immediate term. Beyond the short term, it is my hope that at a higher level, several principles can distilled from the dissertation to inform secure system designs in future. The following research directions capture my vision towards more secure generation of systems.

Security-Aware Hardware-Software Co-Design. Security is indeed a full-system property, and the increasing incidence of microarchitectural-style[1] attacks (such as CLKscrew [146], Rowhammer [79], and more recently, Meltdown [88] and Spectre [83]) bears testament to this. The interesting aspect to these classes of attacks is that the root cause does not lie with any independent hardware or software component, but rather with the way multiple components interact with one another. For example, with CLKscrew, taken in isolation the individual constituents of the energy management stack are relatively well-designed — ARM Trustzone correctly prevents lower-privileged software from directly accessing code and data within the trusted zone; the DVFS drivers enforce configuration limits at the software level; the hardware regulators are designed to operate efficiently across multiple hardware components. However, taking the coexistence of these designs in tandem allows a lower-privileged attacker to influence the operating frequency and voltage of trusted computation from outside Trustzone. While attacks are mainly geared towards the higher layers of the computing stack for the past two decades, the lower layers closer to hardware are becoming more of a target in recent years. In short, to a determined malicious actor, all components in systems are fair game.

Moving forward, I believe we need to rethink how we approach the early life cycle of system designs. First, beyond optimizing independent components solely for performance, functional correctnessness and robustness, it is imperative that security be made a design consideration during the conception, implementation and evaluation of hardware components. Second, since

[1]Compared to *architectural* attacks that target specific OS, ISA or platforms, *microarchitectural* attacks work across architectures and can be difficult to fix.

many microarchitectural attacks result from the inter-component interaction, the conventional approach to designing system components in silos should go. Hardware architects and system integrators should take an active role in carefully examining the interplay amongst different system components. A cursory systems-level brainstorming in the design stage—systematically enumerating all ways system components can function and interact with one another—may uncover previously unanticipated ways users (and attackers) can combine the use of different system components in tandem to nefarious ends. The straightforward approach is to begin with a qualitative approach to assessing the security of hardware designs, perhaps with dedicated security assessment teams adept at conducting hardware-oriented vulnerability analysis. However, a potential obstacle now is the lack of techniques to properly quantify security, or the lack thereof. In large firms involved in the early hardware design, evaluating security of individual components, much less the cross-interaction of multiple components, can prove unwieldy because it is difficult to measure security. Further research into developing concrete quantitative measures of security (both intra-component and inter-component) is likely warranted.

Always-on Security-Oriented Hardware. This dissertation shows even though much semantic information about software execution is lost at the hardware level, useful signals can still be gleaned to detect anomalous software behavior. This suggests that we can possibly build anomaly detection mechanisms directly into hardware. I envision a next-generation security architecture where control-flow and data-flow profiling mechanisms are designed as standard customization elements in system designs. In a defense-in-depth setting, these in-hardware security units can provide the first line of defense in a ensemble deployment of malware/exploitation detection "sensors". These hardware defense sensors can offer protection in two ways: (1) function as a lightweight low-overhead trigger to defer analysis and detection to higher-overhead but more accurate software-based sensors, (2) provide an anomaly score (as a continuous value, instead of a discrete one) that can then be combined with the output scores of other sensors in an ensemble fashion to achieve a better detection performance.

In-Flux Hardware. Attackers prey on predictability of system states. For instance, recent

microarchitectural attacks (`CLKscrew`, Meltdown, Spectre) are enabled in part by predictable timing side channels of residual system states. Advanced dynamic code reuse attacks exploit predictable patterns in executable memory to execute. Mimicry-style malware exploit predictability in models to evade detection. Thus, defenses based on the moving-target principle of removing predictability have proven fairly effective and robust. To push this defense paradigm further, I think the time is ripe to build hardware that can, by design, inject unpredictability or randomness into runtime states.

Hardware have always been designed with predictability of timing and execution states. This determinism can potentially be removed by creating hardware that changes aspects of its microarchitectural characteristics each time a software program executes – *i.e.* the hardware is constantly "in-flux". Sethumadhavan *et al.* coin the term *polymorphic hardware* to describe such a shape-shifting architecture [129]. It makes systems more resilence to attacks in two ways, namely, one, by making it hard to gain initial runtime information to bootstrap attacks, and two, by making any acquired knowledge of runtime information for attacks obsolete quickly. These in-flux architectures assume that vulnerabilities and exploits are inevitable, and shift the defensive posture from impenetrability to resiliency, which arguably may be more sustainable in the long term.

Bibliography

[1] ARM. *Power Management with big.LITTLE: A technical overview*. https://community.arm.com/processors/b/blog/posts/power-management-with-big-little-a-technical-overview. 2013.

[2] ARM. "Security Technology - Building a Secure System using TrustZone Technology". In: *ARM Technical White Paper* (2009).

[3] ARM. *c9, Performance Monitor Control Register*. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344b/Bgbdeggf.html. Cortex-A8 Technical Reference Manual.

[4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow integrity principles, implementations, and applications". In: *ACM Trans. Inf. Syst. Secur.* 13.1 (Nov. 2009), 4:1–4:40. ISSN: 1094-9224. DOI: 10.1145/1609956.1609960. URL: http://doi.acm.org/10.1145/1609956.1609960.

[5] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. "Control-flow integrity". In: *Proceedings of the 12th ACM conference on Computer and communications security*. ACM. 2005, pp. 340–353.

[6] Amazon. *Processor State Control for Your EC2 Instance*. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/processor_state_control.html. Amazon AWS.

[7] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. "Innovative technology for CPU based attestation and sealing". In: *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy (HASP)*. Vol. 13. 2013.

[8] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. "An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries". In: *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, 2016, pp. 583–600.

[9] Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. "The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines". In: *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS 2015)*. 2015.

[10] Michael Backes and Stefan Nürnberger. "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing". In: *Proc. 23rd Usenix Security Sym* (2014), pp. 433–447.

[11] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. "You Can Run but You Can'T Read: Preventing Disclosure Exploits in Executable Code". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, 2014, pp. 1342–1353. ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660378. URL: http://doi.acm.org/10.1145/2660267.2660378.

[12] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. "The sorcerer's apprentice guide to fault attacks". In: *Proceedings of the IEEE* 94.2 (2006), pp. 370–382.

[13] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. "Countermeasures against fault attacks on software implemented AES: effectiveness and cost". In: *Proceedings of the 5th Workshop on Embedded Systems Security*. ACM. 2010, p. 7.

[14] Alessandro Barenghi, Guido Bertoni, Emanuele Parrinello, and Gerardo Pelosi. "Low voltage fault attacks on the RSA cryptosystem". In: *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2009 Workshop on*. IEEE. 2009, pp. 23–31.

[15] Jeff Barr. *Now Available - New C4 Instances*. https://aws.amazon.com/blogs/aws/now-available-new-c4-instances/. Jan. 2015.

[16] Sean Beaupre. *TRUSTNONE - Signed comparison on unsigned user input*. http://theroot.ninja/disclosures/TRUSTNONE_1.0-11282015.pdf.

[17] Gal Beniamini. *Trust Issues: Exploiting TrustZone TEEs*. https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html. July 2017.

[18] Alexandre Berzati, Cécile Canovas, and Louis Goubin. "Perturbating RSA public keys: An improved attack". In: *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer. 2008, pp. 380–395.

[19] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. "Timely rerandomization for mitigating memory disclosures". In: *Proceedings of the 22nd*

ACM SIGSAC Conference on Computer and Communications Security. ACM. 2015, pp. 268–279.

[20] Eli Biham, Yaniv Carmeli, and Adi Shamir. "Bug attacks". In: *Annual International Cryptology Conference*. Springer. 2008, pp. 221–240.

[21] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. "Jump-oriented programming: a new class of code-reuse attack". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM. 2011, pp. 30–40.

[22] Johannes Blömer, Ricardo Gomes Da Silva, Peter Günther, Juliane Krämer, and Jean-Pierre Seifert. "A practical second-order fault attack against a real-world pairing implementation". In: *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2014 Workshop on*. IEEE. 2014, pp. 123–136.

[23] Dan Boneh. "Twenty years of attacks on the RSA cryptosystem". In: *Notices of the American Mathematical Society (AMS)* 46.2 (1999), pp. 203–213.

[24] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. "On the Importance of Checking Cryptographic Protocols for Faults". In: *Proceedings of the 16th Annual International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT'97. Springer-Verlag, 1997, pp. 37–51. ISBN: 3-540-62975-0. URL: http://dl.acm.org/citation.cfm?id=1754542.1754548.

[25] Jean-Marie Borello and Ludovic Mé. "Code obfuscation techniques for metamorphic viruses". In: *Journal in Computer Virology* 4.3 (2008), pp. 211–220.

[26] Yuriy Bulygin. *Attacking and Defending BIOS in 2015*. http://www.intelsecurity.com/advanced-threat-research/content/AttackingAndDefendingBIOS-RECon2015.pdf. 2015.

[27] G. Canivet, P. Maistri, R. Leveugle, J. Clédière, F. Valette, and M. Renaudin. "Glitch and Laser Fault Attacks onto a Secure AES Implementation on a SRAM-Based FPGA". In: *Journal of Cryptology* 24.2 (2011), pp. 247–268. ISSN: 1432-1378. DOI: 10.1007/s00145-010-9083-9. URL: http://dx.doi.org/10.1007/s00145-010-9083-9.

[28] Miguel Castro, Manuel Costa, and Tim Harris. "Securing software by enforcing data-flow integrity". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, pp. 147–160.

[29] Haibo Chen, Liwei Yuan, Xi Wu, Binyu Zang, Bo Huang, and Pen-Chung Yew. "Control flow obfuscation with information flow tracking". In: *MICRO*. 2009, pp. 391–400.

[30] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. "Non-control-data Attacks Are Realistic Threats". In: *Proceedings of the 14th Conference on USENIX Secu-*

rity Symposium - Volume 14. SSYM'05. USENIX Association, 2005, pp. 12–12. URL: http://dl.acm.org/citation.cfm?id=1251398.1251410.

[31] Shane S Clark, Benjamin Ransford, Amir Rahmati, Shane Guineau, Jacob Sorber, Kevin Fu, and Wenyuan Xu. "WattsUpDoc: Power Side Channels to Nonintrusively Discover Untargeted Malware on Embedded Medical Devices". In: *USENIX Workshop on Health Information Technologies*. Aug. 2013.

[32] Stephen J Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. "It's a TRaP: Table randomization and protection against function-reuse attacks". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2015, pp. 243–255.

[33] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. "Readactor: Practical Code Randomization Resilient to Memory Disclosure". In: *36th IEEE Symposium on Security and Privacy (Oakland)*. May 2015.

[34] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. "Thwarting cache side-channel attacks through dynamic software diversity". In: *Network And Distributed System Security Symposium, NDSS*. Vol. 15. 2015.

[35] Ang Cui, Michael Costello, and Salvatore J Stolfo. "When Firmware Modifications Attack: A Case Study of Embedded Exploitation." In: *NDSS*. 2013.

[36] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming". In: *Proc. 22nd Network and Distributed Systems Security Sym.(NDSS)* (2015).

[37] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. "On the feasibility of online malware detection with performance counters". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA '13. ACM, 2013, pp. 559–570. ISBN: 978-1-4503-2079-5. DOI: 10.1145/2485922.2485970.

[38] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. "SPIDER: Stealthy Binary Program Instrumentation and Debugging via Hardware Virtualization". In: *Proceedings of the 29th Annual Computer Security Applications Conference*. ACSAC '13. New Orleans, Louisiana, USA: ACM, 2013, pp. 289–298. ISBN: 978-1-4503-2015-3.

[39] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Victor Lomné, and Florian Mendel. "Statistical Fault Attacks on Nonce-Based Authenticated Encryption Schemes". In: *Advances in Cryptology – ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8,*

2016, Proceedings, Part I. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 369–395. ISBN: 978-3-662-53887-6. DOI: 10.1007/978-3-662-53887-6_14. URL: http://dx.doi.org/10.1007/978-3-662-53887-6_14.

[40] R O Duda, P E Hart, and D G Stork. "Pattern Classification, New York: John Wiley & Sons, 2001, pp. xx + 654". In: *J. Classif.* 24.2 (Sept. 2007), pp. 305–307. ISSN: 0176-4268. DOI: 10.1007/s00357-007-0015-9.

[41] Haakon Dybdahl, Per Gunnar Kjeldsberg, Marius Grannæs, and Lasse Natvig. "Destructive-read in Embedded DRAM, Impact on Power Consumption". In: *J. Embedded Comput.* 2.2 (Apr. 2006), pp. 249–260. ISSN: 1740-4460. URL: http://dl.acm.org/citation.cfm?id=1370998.1371000.

[42] Jake Edge. *KS2012: ARM: Secure monitor API*. https://lwn.net/Articles/513756/. Aug. 2012.

[43] Jan-Erik Ekberg and Kari Kostiainen. *Trusted Execution Environments on Mobile Devices*. https://www.cs.helsinki.fi/group/secures/CCS-tutorial/tutorial-slides.pdf. ACM CCS 2013 tutorial. Nov. 2013.

[44] Isaac Evans, Sam Fingeret, Julián González, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. "Missing the Point(er): On the Effectiveness of Code Pointer Integrity". In: *36th IEEE Symposium on Security and Privacy (Oakland)*. May 2015.

[45] Stephen Fewer. *Reflective DLL Injection*. http://www.harmonysecurity.com/files/HS-P005_ReflectiveDllInjection.pdf. Oct. 2008.

[46] *Firmware update for Nexus 6 (shamu)*. https://dl.google.com/dl/android/aosp/shamu-mob31s-factory-c73a35ef.zip. Factory Images for Nexus and Pixel Devices.

[47] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. "A sense of self for unix processes". In: *1996 IEEE Symposium on Security and Privacy*. 1996, pp. 120–128.

[48] Jeff Forristal. *Hardware Involved Software Attacks*. http://forristal.com/material/Forristal_Hardware_Involved_Software_Attacks.pdf. 2012.

[49] Fyyre. *Disable PatchGuard - the easy/lazy way*. http://fyyre.ivory-tower.de/projects/bootloader.txt. 2011.

[50] Tal Garfinkel, Mendel Rosenblum, et al. "A Virtual Machine Introspection Based Architecture for Intrusion Detection." In: *NDSS*. Vol. 3. 2003, pp. 191–206.

[51] Jason Gionta, William Enck, and Peng Ning. "HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities". In: *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. CODASPY '15. 2015, pp. 325–336.

[52] Carlos R Aguayo Gonzalez and Jeffrey H Reed. "Detecting unauthorized software execution in SDR using power fingerprinting". In: *MILITARY COMMUNICATIONS CONFERENCE, 2010-MILCOM 2010*. IEEE. 2010, pp. 2211–2216.

[53] Google. *Multiplatform Content Protection for Internet Video Delivery*. `https://www.widevine.com/wv_drm.html`. Widevine DRM.

[54] Sudhakar Govindavajhala and Andrew W. Appel. "Using Memory Errors to Attack a Virtual Machine". In: *Proceedings of the 2003 IEEE Symposium on Security and Privacy (S&P)*, pp. 154–165.

[55] Michael Gruhn and Tim Muller. "On the practicability of cold boot attacks". In: *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*. IEEE. 2013, pp. 390–397.

[56] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer. js: A remote software-induced fault attack in javascript". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 300–321.

[57] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches". In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 897–912.

[58] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. "Lest we remember: cold-boot attacks on encryption keys". In: *Communications of the ACM* 52.5 (2009), pp. 91–98.

[59] Per Hammarlund, Rajesh Kumar, Randy B Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stephan Jourdan, et al. "Haswell: The fourth-generation Intel core processor". In: *IEEE Micro* 2 (2014), pp. 6–20.

[60] Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*. Vol. 21. 2. ACM, 1993.

[61] Johannes Hoffmann, Stephan Neumann, and Thorsten Holz. "Mobile Malware Detection Based on Energy Fingerprints: A Dead End?" In: *Research in Attacks, Intrusions, and Defenses*. Springer, 2013, pp. 348–368.

[62] Steven A Hofmeyr, Stephanie Forrest, and Anil Somayaji. "Intrusion detection using sequences of system calls". In: *Journal of computer security* 6.3 (1998), pp. 151–180.

[63] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. "Librando: transparent code randomization for just-in-time compilers". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*. ACM. 2013, pp. 993–1004.

[64] K. Hoste and L. Eeckhout. "Comparing Benchmarks Using Key Microarchitecture-Independent Characteristics". In: *Workload Characterization, 2006 IEEE International Symposium on*. IEEE, Oct. 2006, pp. 83–92. ISBN: 1-4244-0508-4. DOI: 10.1109/iiswc.2006.302732.

[65] Nguyen Minh Huu, Bruno Robisson, Michel Agoyan, and Nathalie Drach. "Low-cost recovery for the code integrity protection in secure embedded processors". In: *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*. IEEE. 2011, pp. 99–104.

[66] Intel. *Behind Intel's New Random-Number Generator*. https://spectrum.ieee.org/computing/hardware/behind-intels-new-randomnumber-generator. Aug. 2011.

[67] Intel. *Control-flow Enforcement Technology Preview*. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcementtechnology-preview.pdf. June 2016.

[68] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3C*. 2014.

[69] Intel. *Intel Software Guard Extensions (Intel SGX)*. https://software.intel.com/en-us/sgx.

[70] Intel. *Intel Transactional Synchronization Extensions (Intel TSX) Overview*. https://software.intel.com/en-us/node/524022.

[71] Intel. *Intel® Advanced Encryption Standard Instructions (AES-NI)*. https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni. Feb. 2012.

[72] Intel. *The Engine for Digital Transformation in the Data Center*. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-e5-brief.pdf. Intel Product Brief.

[73] Brian Jeff. "big.LITTLE system architecture from arm: Saving power through heterogeneous multiprocessing and task context migration". In: *Proceedings of the 49th Annual Design Automation Conference (DAC)*. ACM. 2012, pp. 1143–1146.

[74] Mehmet Kayaalp, Timothy Schmitt, Junaid Nomani, Dmitry Ponomarev, and Nael B Abu-Ghazaleh. "SCRAP: Architecture for signature-based protection from Code Reuse Attacks." In: *HPCA*. 2013, pp. 258–269.

[75] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. "Countering code-injection attacks with instruction-set randomization". In: *Proceedings of the 10th ACM conference on Computer and communications security*. ACM. 2003, pp. 272–280.

[76] Chongkyung Kil, Jinsuk Jim, Christopher Bookholt, Jun Xu, and Peng Ning. "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software". In: *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE. 2006, pp. 339–348.

[77] Hahnsang Kim, Joshua Smith, and Kang G. Shin. "Detecting Energy-greedy Anomalies and Mobile Malware Variants". In: *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*. MobiSys '08. ACM, 2008. DOI: 10.1145/1378600.1378627. URL: http://doi.acm.org/10.1145/1378600.1378627.

[78] Hahnsang Kim, Joshua Smith, and Kang G Shin. "Detecting energy-greedy anomalies and mobile malware variants". In: *Proceedings of the 6th international conference on Mobile systems, applications, and services*. ACM. 2008, pp. 239–252.

[79] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors". In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. June 2014, pp. 361–372.

[80] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors". In: *Proceeding of the 41st annual international symposium on Computer architecuture*. IEEE Press. 2014, pp. 361–372.

[81] Cetin Kaya Koc. *High-speed RSA implementation*. Tech. rep. Technical Report, RSA Laboratories, 1994.

[82] Paul C Kocher. "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems". In: *Advances in Cryptology - CRYPTO'96*. Springer. 1996, pp. 104–113.

[83] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *ArXiv e-prints* (Jan. 2018). arXiv: 1801.01203.

[84] Deguang Kong, Donghai Tian, Peng Liu, and Dinghao Wu. "SA3: Automatic semantic aware attribution analysis of remote exploits". In: *Security and Privacy in Communication Networks*. Springer, 2012, pp. 190–208.

[85] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. "Code-Pointer Integrity". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 2014.

[86] Hendrik W Lenstra Jr. "Factoring integers with elliptic curves". In: *Annals of mathematics* (1987), pp. 649–673.

[87] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clementine Maurice, and Stefan Mangard. "ARMageddon: Cache Attacks on Mobile Devices". In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX, 2016, pp. 549–564.

[88] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown". In: *ArXiv e-prints* (Jan. 2018). arXiv: 1801.01207.

[89] ARM Ltd. *ARM Trustzone*. https://www.arm.com/products/security-on-arm/trustzone.

[90] Pei Luo, Chao Luo, and Yunsi Fei. *System Clock and Power Supply Cross-Checking for Glitch Detection*. Cryptology ePrint Archive, Report 2016/968. http://eprint.iacr.org/2016/968. 2016.

[91] Dan Luu. *We saw some really bad Intel CPU bugs in 2015, and we should expect to see more in the future*. http://danluu.com/cpu-bugs/. Jan. 2016.

[92] MITRE. *Heartbleed - CVE-2014-0160*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160. 2014.

[93] *MSM Subsystem Power Manager (spm-v2)*. https://android.googlesource.com/kernel/msm.git/+/android-msm-shamu-3.10-lollipop-mr1/Documentation/devicetree/bindings/arm/msm/spm-v2.txt. Git at Google.

[94] Matthew V Mahoney. "Network traffic anomaly detection based on packet bytes". In: *Proceedings of the 2003 ACM symposium on Applied computing*. 2003, pp. 346–350.

[95] Corey Malone, Mohamed Zahran, and Ramesh Karri. "Are hardware performance counters a cost effective way for integrity checking of programs". In: *Proceedings of the sixth ACM workshop on Scalable trusted computing*. STC '11. Chicago, Illinois, USA: ACM, 2011, pp. 71–76. ISBN: 978-1-4503-1001-7. DOI: 10.1145/2046582.2046596.

[96] Carla Marceau. "Characterizing the behavior of a program using multiple-length n-grams". In: *Proceedings of the 2000 workshop on New security paradigms*. ACM. 2001, pp. 101–110.

[97] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 1996. ISBN: 0849385237.

[98] Microsoft. *Asynchronous Procedure Calls*. `https://msdn.microsoft.com/en-us/library/windows/desktop/ms681951(v=vs.85).aspx`.

[99] Microsoft. *Windows Resource Protection*. `https://msdn.microsoft.com/en-us/library/windows/desktop/cc185681(v=vs.85).aspx`.

[100] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Advanced Encryption Standard*. Alpha Press, 2009. ISBN: 6130268297, 9786130268299.

[101] *Mobile Hardware Stats 2016-09*. `http://hwstats.unity3d.com/mobile/cpu.html`. Unity. Sept. 2016.

[102] Peter L Montgomery. "Modular multiplication without trial division". In: *Mathematics of computation* 44.170 (1985), pp. 519–521.

[103] Micah Morton, Hyungjoon Koo, Forrest Li, Kevin Z. Snow, Michalis Polychronakis, and Fabian Monrose. "Defeating Zombie Gadgets by Re-randomizing Code upon Disclosure". In: *Engineering Secure Software and Systems: 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings*. Springer International Publishing, 2017, pp. 143–160.

[104] *Nexus 6 Qualcomm-stipulated OPP*. `https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-lollipop-mr1/arch/arm/boot/dts/qcom/apq8084.dtsi`. Git at Google.

[105] Colin O'Flynn. *Fault Injection using Crowbars on Embedded Systems*. Tech. rep. IACR Cryptology ePrint Archive, 2016.

[106] Venkatesh Pallipadi and Alexey Starikovskiy. "The ondemand governor". In: *Proceedings of the Linux Symposium*. Vol. 2. 00216. sn. 2006, pp. 215–230.

[107] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization". In: *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE. 2012, pp. 601–615.

[108] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. "Transparent ROP exploit mitigation using indirect branch tracing". In: *Proceedings of the 22nd USENIX conference on Security (USENIX '13)*. USENIX Association, 2013, pp. 447–462.

[109] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990. ISBN: 1-55880-069-8.

[110] Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo. “Analysis of computer intrusions using sequences of function calls”. In: *Dependable and Secure Computing, IEEE Transactions on* 4.2 (2007), pp. 137–150.

[111] Gabor Pek, Andrea Lanzi, Abhinav Srivastava, Davide Balzarotti, Aurelien Francillon, and Christoph Neumann. “On the feasibility of software attacks on commodity virtual machine monitors via direct device assignment”. In: *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM. 2014, pp. 305–316.

[112] Jannik Pewny, Philipp Koppe, Lucas Davi, and Thorsten Holz. “Breaking and Fixing Destructive Code Read Defenses”. In: *Proceedings of 33nd Annual Computer Security Applications Conference (ACSAC)*. 2017.

[113] Pinkie Pie. *Mobile Pwn2Own Autumn 2013 - Chrome on Android - Exploit Writeup*. 2013.

[114] Michalis Polychronakis, Kostas G Anagnostakis, and Evangelos P Markatos. “Comprehensive shellcode detection using runtime heuristics”. In: *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM. 2010, pp. 287–296.

[115] Michalis Polychronakis, Kostas G Anagnostakis, and Evangelos P Markatos. “Emulation-based detection of non-self-contained polymorphic shellcode”. In: *Recent Advances in Intrusion Detection*. Springer. 2007, pp. 87–106.

[116] *QSEECOM source code*. `https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-lollipop-mr1/drivers/misc/qseecom.c`. Git at Google.

[117] *Qualcomm Krait PMIC frequency driver source code*. `https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-lollipop-mr1/drivers/clk/qcom/clock-krait.c`. Git at Google.

[118] *Qualcomm Krait PMIC voltage regulator driver source code*. `https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-lollipop-mr1/arch/arm/mach-msm/krait-regulator.c`. Git at Google.

[119] Qualcomm. *Qualcomm Announces Breakthrough Mobile Anti-malware Technology Utilizing Cognitive Computing*. `https://www.qualcomm.com/news/releases/2015/08/31/qualcomm-announces-breakthrough-mobile-anti-malware-technology-utilizing`. Aug. 2015.

[120] Qualcomm. *Secure Boot and Image Authentication - Technical Overview*. `https://www.qualcomm.com/documents/secure-boot-and-image-authentication-technical-overview`. Oct. 2016.

[121] Qualcomm. *Snapdragon S4 Processors: System on Chip Solutions for a New Mobile Age*. https://www.qualcomm.com/documents/snapdragon-s4-processors-system-chip-soluti
ons-new-mobile-age. July 2013.

[122] Qualcomm. *Whitepaper: Pointer Authentication on ARMv8.3*. https://www.qualcomm.com/documents/whitepaper-pointer-authentication-armv83. Jan. 2017.

[123] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. "Flip Feng Shui: Hammering a Needle in the Software Stack". In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX, 2016, pp. 1–18.

[124] Ronald L Rivest, Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[125] STMicroelectronics. *E-fuses*. http://www.st.com/en/power-management/e-fuses.html?querycriteria=productId=SC1532. How-swap power management.

[126] Mark Seaborn and Thomas Dullien. "Exploiting the DRAM rowhammer bug to gain kernel privileges". In: *Black Hat* (2015).

[127] Jeff Seibert, Hamed Okkhravi, and Eric Söderström. "Information leaks without memory disclosures: Remote side channel attacks on diversified code". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2014, pp. 54–65.

[128] R Sekar, Mugdha Bendre, Dinakar Dhurjati, and Pradeep Bollineni. "A fast automaton-based method for detecting anomalous program behaviors". In: *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*. IEEE. 2001, pp. 144–155.

[129] Simha Sethumadhavan, Salvatore J Stolfo, Angelos Keromytis, Junfeng Yang, and David August. "The sparchs project: Hardware support for software security". In: *SysSec Workshop (SysSec), 2011 First*. IEEE. 2011, pp. 119–122.

[130] Hovav Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)". In: *Proceedings of the 14th ACM conference on Computer and communications security*. ACM. 2007, pp. 552–561.

[131] Findlay Shearer. *Power Management in Mobile Devices*. Newnes, 2011.

[132] Kai Shen, Ming Zhong, Sandhya Dwarkadas, Chuanpeng Li, Christopher Stewart, and Xiao Zhang. "Hardware counter driven on-the-fly request signatures". In: *Proceedings of the 13th international conference on Architectural support for programming languages and*

operating systems. ASPLOS XIII. ACM, 2008, pp. 189–200. ISBN: 978-1-59593-958-6. DOI: 10.1145/1346281.1346306.

[133] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization". In: *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE. 2013, pp. 574–588.

[134] Kevin Z Snow, Roman Rogowski, Jan Werner, Hyungjoon Koo, Fabian Monrose, and Michalis Polychronakis. "Return to the zombie gadgets: Undermining destructive code reads via code inference attacks". In: *Proceedings of the 2016 IEEE Symposium on Security and Privacy*. IEEE. 2016.

[135] Anil Somayaji and Stephanie Forrest. "Automated response using system-call delays". In: *Proceedings of the 9th USENIX Security Symposium*. Vol. 70. 2000.

[136] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. "Exploiting and protecting dynamic code generation". In: *Proceedings of the 2015 Network and Distributed System Security (NDSS) Symposium*. 2015.

[137] Sherri Sparks and Jamie Butler. *Raising The Bar For Windows Rootkit Detection*. http://phrack.org/issues/63/8.html. 2005.

[138] Brinkley Sprunt. "Pentium 4 performance-monitoring features". In: *Ieee Micro* 22.4 (2002), pp. 72–82.

[139] Z Stamenković, V Petrović, and G Schoof. "Fault-tolerant ASIC: Design and implementation". In: *Facta universitatis-series: Electronics and Energetics* 26.3 (2013), pp. 175–186.

[140] Patrick Stewin. "A Primitive for Revealing Stealthy Peripheral-Based Attacks on the Computing Platform's Main Memory". In: *Research in Attacks, Intrusions, and Defenses*. Springer, 2013, pp. 1–20.

[141] Patrick Stewin and Iurii Bystrov. "Understanding DMA malware". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2013, pp. 21–41.

[142] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. "Breaking the memory secrecy assumption". In: *Proceedings of the Second European Workshop on System Security*. ACM. 2009, pp. 1–8.

[143] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal War in Memory". In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 48–62. ISBN: 978-0-7695-4977-4. DOI: 10.1109/SP.2013.13. URL: http://dx.doi.org/10.1109/SP.2013.13.

[144] Peter Szor. *The art of computer virus research and defense*. Pearson Education, 2005.

[145] Adrian Tang, Simha Sethumadhavan, and Salvatore J Stolfo. "Unsupervised anomaly-based malware detection using hardware features". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2014, pp. 109–129.

[146] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management". In: *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 2017, pp. 1057–1074.

[147] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2015, pp. 256–267.

[148] PaX Team. *PaX address space layout randomization (ASLR)*. 2003.

[149] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. "Architectural Support for Copy and Tamper Resistant Software". In: *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IX. Cambridge, Massachusetts, USA: ACM, 2000, pp. 168–177. ISBN: 1-58113-317-0. DOI: 10.1145/378993.379237. URL: http://doi.acm.org/10.1145/378993.379237.

[150] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. "Architectural Support for Copy and Tamper Resistant Software". In: *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IX. Cambridge, Massachusetts, USA: ACM, 2000, pp. 168–177. ISBN: 1-58113-317-0. DOI: 10.1145/378993.379237. URL: http://doi.acm.org/10.1145/378993.379237.

[151] Jacob Torrey. *More shadow walker: Tlb-splitting on modern x86*. Blackhat USA. 2014.

[152] Xeno Kovah Trammell Hudson and Corey Kallenberg. *Thunderstrike 2: Sith Strike - A MacBook firmware worm*. https://www.blackhat.com/docs/us-15/materials/us-15-Hudson-Thunderstrike-2-Sith-Strike.pdf. Aug. 2015.

[153] TrendMicro. *The Crimeware Evolution (Research Whitepaper)*. 2012.

[154] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. "Differential Fault Analysis of the Advanced Encryption Standard using a Single Fault". In: *IFIP International Workshop on Information Security Theory and Practices*. Springer. 2011, pp. 224–233.

[155] P.C. Van Oorschot, A. Somayaji, and G. Wurster. "Hardware-assisted circumvention of self-hashing software tamper resistance". In: *Dependable and Secure Computing, IEEE Transactions on* 2.2 (2005), pp. 82–92. ISSN: 1545-5971. DOI: 10.1109/TDSC.2005.24.

[156] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Nov. 2016. URL: https://vvdveen.com/publications/drammer.pdf.

[157] Rajesh Velegalati, Kinjal Shah, and Jens-Peter Kaps. "Glitch Detection in Hardware Implementations on FPGAs Using Delay Based Sampling Techniques". In: *Proceedings of the 2013 Euromicro Conference on Digital System Design*. DSD '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 947–954. ISBN: 978-1-4799-2978-8. DOI: 10.1109/DSD.2013.107. URL: http://dx.doi.org/10.1109/DSD.2013.107.

[158] Bo Wang, Leibo Liu, Chenchen Deng, Min Zhu, Shouyi Yin, and Shaojun Wei. "Against Double Fault Attacks: Injection Effort Model, Space and Time Randomization Based Countermeasures for Reconfigurable Array Architecture". In: *IEEE Transactions on Information Forensics and Security* 11.6 (2016), pp. 1151–1164.

[159] Ke Wang, Janak J Parekh, and Salvatore J Stolfo. "Anagram: A content anomaly detector resistant to mimicry attack". In: *Recent Advances in Intrusion Detection*. Springer. 2006, pp. 226–248.

[160] Xueyang Wang and Ramesh Karri. "NumChecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters". In: *Proceedings of the 50th Annual Design Automation Conference*. DAC '13. Austin, Texas: ACM, 2013, 79:1–79:7. ISBN: 978-1-4503-2071-9. DOI: 10.1145/2463209.2488831.

[161] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 157–168.

[162] Richard Wartell, Yan Zhou, Kevin W Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. "Differentiating code from data in x86 binaries". In: *Machine Learning and Knowledge Discovery in Databases*. Springer, 2011, pp. 522–536.

[163] Vincent M Weaver. *Advanced Hardware Profiling and Sampling (PEBS, IBS, etc.): Creating a New PAPI Sampling Interface*. Aug. 2016.

[164] Vincent M Weaver and Sally A McKee. "Can hardware performance counters be trusted?" In: *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE. 2008, pp. 141–150.

[165] Vincent M Weaver, Dan Terpstra, and Shirley Moore. "Non-determinism and overcount on modern hardware performance counter implementations". In: *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE. 2013, pp. 215–224.

[166] Ralf-Philipp Weinmann. *Concurrency: A problem and opportunity in the exploitation of memory corruptions*. `https://cansecwest.com/slides/2014/rpw-csw2014-merged.pdf`. 2014.

[167] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. "Scheduling for Reduced CPU Energy". In: *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 1994.

[168] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z Snow, Fabian Monrose, and Michalis Polychronakis. "No-execute-after-read: Preventing code disclosure in commodity software". In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM. 2016, pp. 35–46.

[169] Wikipedia. *Shellcode*. `https://en.wikipedia.org/wiki/Shellcode`.

[170] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. "Shuffler: Fast and Deployable Continuous Code Re-Randomization". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016, pp. 367–382.

[171] Rafal Wojtczuk. "Poacher turned gamekeeper: Lessons learned from eight years of breaking hypervisors". In: *Black Hat*. 2014.

[172] Rafal Wojtczuk and Corey Kallenberg. *Attacks on UEFI Security*. `https://bromiumlabs.files.wordpress.com/2015/01/attacksonuefi_slides.pdf`. 2015.

[173] Rafal Wojtczuk and Alexander Tereshkin. *Attacking Intel BIOS*. `http://www.blackhat.com/presentations/bh-usa-09/WOJTCZUK/BHUSA09-Wojtczuk-AtkIntelBios-SLIDES.pdf`. Aug. 2009.

[174] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. "CFIMon: Detecting violation of control flow integrity using performance counters". In: *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. DSN '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–12. ISBN: 978-1-4673-1624-8.

[175] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, pp. 719–732. ISBN: 978-1-931971-15-7.

[176] Liwei Yuan, Weichao Xing, Haibo Chen, and Binyu Zang. "Security breaches as PMU deviation: detecting and identifying security attacks using performance counters". In: *APSys*. 2011, p. 6.

[177] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. “Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2016.

[178] btarunr. *Rejoice! Base Clock Overclocking to Make a Comeback with Skylake*. `https://www.techpowerup.com/218315/rejoice-base-clock-overclocking-to-make-a-comeback-with-skylake`. TechPowerup. 2015.

Appendix

A.1 Example Glitch in RSA Modulus

Original Modulus N:

```
...f35a...
```

Corrupted Modulus N_A:

```
c44dc735f6682a261a0b8545a62dd13df4c646a5ede482cef85892 5baa1811fa0284766b3d1d2b4d6893df4d9c045efe3e84d8c5d036
31b25420f1231d8211e2322eb7eb524da6c1e8fb4c3ae4a8f5ca13 d1e0591f5c64e8e711b3726215cec59ed0ebc6bb042b917d445288
87915fdf764df691d183e16f31ba1ed94c84b476e74b488463e855 51022021763a3a3a64ddf105c1530ef3fcf7e54233e5d3a4747bbb
17328a63e6e3384ac25ee80054bd566855e2eb59a2fd168d3643e4 4851acf0d118fb03c73ebc099b4add59c39367d6c91f498d8d607a
f2e57cc73e3b5718435a81123f080267726a2a9c1cc94b9c6bb681 7427b85d8c670f9a53a777511b
```

Factors of N_A:

```
0x3, 0x11b, 0xcb9, 0x4a70807d6567959438227805b12a19a73 4365bd998c6a6f7dfd595360ed3bae4d765170d5afb7f425fddb21
91c3f902c5d049dc745b339e884a601e17e081f2faca0b2ea70e64 8b09176143a9ff745a46497b1b30fc8b378ac7d05f46eaceb41b99
47ffca9a810d4e80baf2f3b03236ab6f243de50976d91eaeb25cc3 9b083c796bd34b66e6c3a0c65c26a30e447cda7b51c556b1842ea6
86e148dfa521bbd1fea2357ad7d151511979fea097c92e4a75b707 f6525a020eca181b1976f31f408547c9557f6b7cd74334147a5b41
ee70a8abf377dd5ba6d85cefa9edaf9af2052f403669675464ed3d 1cff75000d2f33ef0d31124b88f83b5690ae3a3883
```

Public Exponent e:

```
0x10001
```

Private Exponent d_A:

```
04160eecc648a3da19abdc42af4cfb41a798e5eb8b1b49c2c29...
```

A.2 Deep Dive into Intel Power Management Controls

In this Appendix section, we explore the power management (PM) mechanisms at the architectural layer of a recent commodity processor – the Intel Haswell processor [59]. Two key advances in this processor, the fully integrated voltage regulator and additional independent voltage/frequency domains, open up new degrees of freedom for the management of dynamic power. We highlight in detail various architectural "knobs" (including some undocumented ones) that are exposed to the software layer for finer-grained power management.

A.2.1 Preliminaries

We perform our walk-through and any empirical measurements using an Intel Core i7-4790k Haswell Quad-Core Desktop processor[2] and Z97 chipset-based motherboard, running Linux Debian 7 OS. While we demonstrate the variety of PM features using Linux, these Haswell-specific features are agnostic to the OS used, and should be available on other OSes such as Windows. Many architectural PM "knobs" exposed to the software can be interrogated and controlled using readily available tools. To aid in our exploration, we leverage two open-sourced tools, which we can install by running the following:

```
$ sudo apt-get install msr-tools devmem2
```

Additionally, we provide code snippets in cases where these tools are insufficient to show the features.

A.2.2 Review of Existing Intel PM Technologies

Before we delve into the newer features on the Haswell processor, we first briefly review two existing Intel technologies pertaining to power management. For each technology we detail its motivation, working mechanisms and ways to view its settings and control these mechanisms from software. We also include snippets of shell commands for readers who want to experience first-hand specific concepts related to each technology. We wrap up this section by showing some empirical measurements of frequency and voltage scaling that are enabled with these two technologies.

A.2.2.1 Enhanced Intel SpeedStep Technology (EIST)

EIST is a processor technology that enables software to dynamically manage the power consumption of a processor (when the processor is actively executing instructions in the C0 state) by initiating performance state, *a.k.a.* P-states, changes. At different P-states, EIST dynamically changes the operating frequency and voltage of the processor at the hardware level. Concealing the gory details of the hardware-level scaling mechanisms from software, EIST empowers software with the ability to control underlying performance states via Model-Specific Register (MSR). These MSRs are special x86-specific control registers that can be read from and written to using the privileged `rdmsr` and `wrmsr` instructions.

[2] Advertised to operate at a base frequency of 4GHz up to the max turbo frequency of 4.4GHz

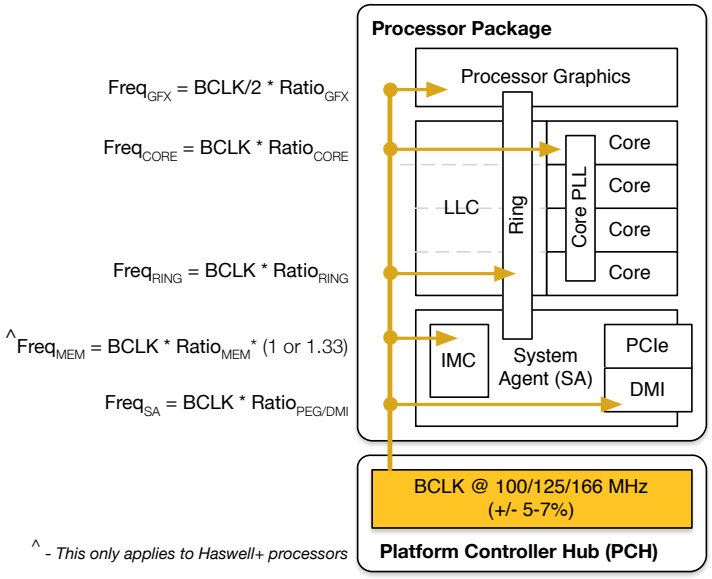

Figure 6.1.1: Intel clock distribution tree.

Before we discuss how software can control frequency changes of the processor, we back up a moment to review how operating frequencies at the microarchitectural level are determined. Beginning from Nehalem processors, Intel replaces the Front-side Bus (FSB), an I/O bus, with the more efficient QuickPath Interconnect (QPI). This introduces a revised clock distribution tree, where the frequencies of various microarchitectural components are derived from a main Base Clock (BCLK) frequency at varying multiplier ratios. Serving as a master clock of sorts on the Intel platform, this BCLK frequency affects the operating frequencies of the cores, DDR memory, the uncore System Agent (SA) (— need to verify the part on SA and recraft this —), the integrated graphics controller and more recently in the Haswell processor, the ring interconnect.

Figure 6.1.1 displays a visual summary of how the frequencies of various microarchitectural components are drawn from the BCLK. At a broad level, if we increase the BCLK, all the frequencies of various components will increase correspondingly. For finer-grained control over frequencies of individual components, we modify the respective multiplier ratios via various controls exposed to software.

EIST allows P-state changes based on only the selection of the frequency multiplier ratio, $Ratio_{CORE}$. In the default mode, the operating voltage will be automatically optimized and configured by PCU. In later sections, we shall see how the processor frequency and voltage can be controlled independently on the newer Haswell processor.

To initiate a change to a desired P-state, software can select the P-state based on $Ratio_{CORE}$, by writing this value into the `IA32_PERF_CTL` $MSR_{0x199}[15:8]$[3]. For example, using `msr-tools`, we can initiate a change to the P-state with $Ratio_{CORE} = 16$ using the following shell command:

```
# wrmsr 0x199 `echo $((16 << 8))`
```

[3]For brevity, we refer to the bit offsets of bl to bh (inclusive) of the MSR of index x as $MSR_x[bh{:}bl]$. If only one bit is used, we use only the bl offset as $MSR_x[bl]$.

EIST offers a feedback mechanism to software to read the currently configured P-state ratio, $Ratio_{CORE}$, from another designated MSR, the `IA32_PERF_STATUS` MSR_{0x198}[15:8]. With this ratio, we can derive the current operating frequency by multiplying this ratio with the BCLK frequency. We read the current P-state ratio as follows:

```
# rdmsr -df 15:8 0x198
16
```

To disable EIST, we can zero out the bit at the `IA32_MISC_ENABLE` MSR_{0x1A0}[16], in which case any P-state change request via further writes to MSR_{0x198}[15:8] will be ignored. Likewise, to know if EIST mode is enabled on a system, we can read the same bit, where a value of 1 indicates that EIST is enabled.

```
# rdmsr -f 16:16 0x1a0
1
```

So far, we have discussed three MSRs related to power management. A noteworthy point is that all these (and subsequently discussed) MSRs exist on a per-core basis, unless otherwise stated. Each logical core has its own set of MSRs.

However, even so, on current Intel architecture, the cores on multi-core processors share the same frequency (and voltage) if they reside in the same processor package. This implies that P-state changes affect all cores at the package level[4]. We can easily verify this by registering a new P-state on one core and reading the measured P-state on another. The new P-state ratio will be reflected in the second core (and the rest of the cores in the same processor package). We can read and configure different values from these MSRs for different cores, using the `-p` flag on the `rdmsr` and `wrmsr` command-line tools, as follows[5]:

```
# rdmsr -p 1 -df 15:8 0x198
8
# wrmsr -p 0 0x199 `echo $((16 << 8))`
# rdmsr -p 1 -df 15:8 0x198
16
```

A.2.2.2 Intel Turbo Boost 2.0

Intel Turbo Boost 2.0 is a processor technology that dynamically increases the frequencies of active processor cores above the Thermal Design Power (TDP) limits for short periods of time to give a temporary "boost" to system performance. The key insight to this technology is the observation that not all the processor cores are active (in C0 or C1 state) at the same time when a system is running certain workloads - some cores in the same processor package could be inactive (in C3 or C6 state). This technology harnesses the additional power leeway afforded by the inactive cores in the form of increased frequency of the active cores to reap the benefits of higher burst performance while staying within the power and thermal limits.

[4]Unlike P-states, C-states can be managed at the granularity of per-physical core basis.

[5]The default *ondemand* CPUfreq governor will make P-state changes dynamically. To ensure the P-state changes that we make in this code snippet remain in effect, we have to use the *userspace* CPUfreq governor instead.

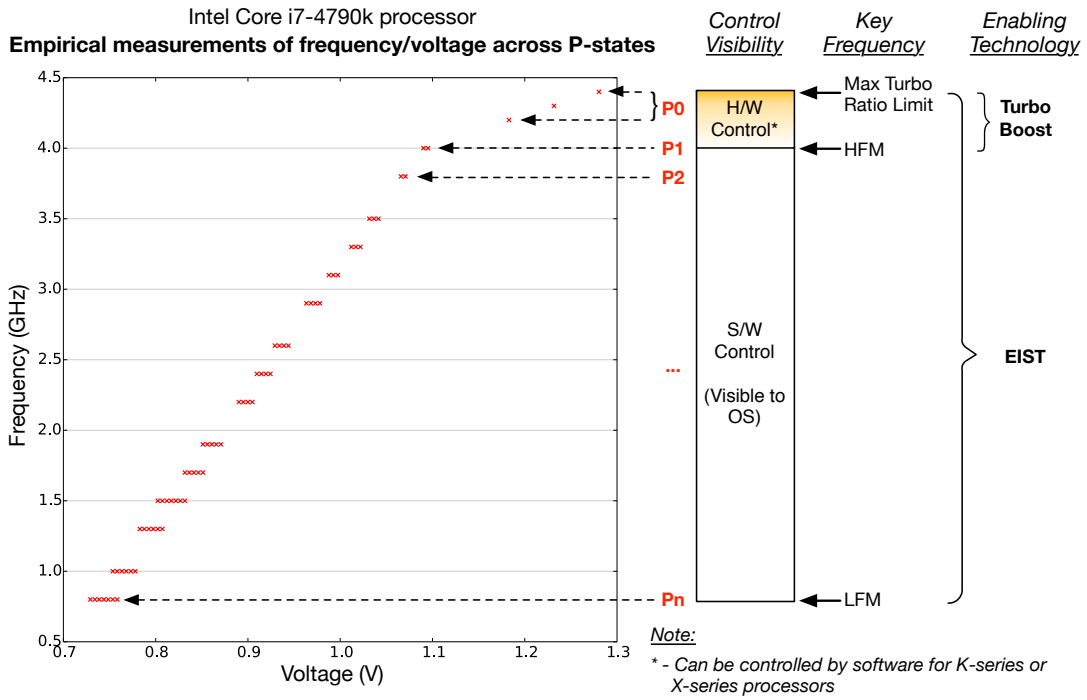

Figure 6.1.2: Empirical measurements of combinations of frequency and voltages across different P-states.

The temporary and opportunistic boost in frequency is primarily hardware-managed by the PCU. The instantaenous core frequency and the length of time the cores stay in the Turbo Boost mode are dependent on a number of factors, including the following:

- **CPU specification**: Frequency limits vary for different CPU SKUs.
- **Number of active cores**: The type of workload determines the number of cores used. The fewer the active cores, the higher the Turbo frequencies achievable.
- **Estimated current and power consumption**: Frequencies can be raised subject to these physical limits.
- **Processor temperature**: Frequencies can only be raised subject to a safe thermal envelope.

When not in Turbo Boost mode, every processor has a stipulated hard limit on the minimum and maximum frequency a core can reach. These frequencies, referred to respectively as Low Frequency Mode (LFM) and High Frequency Mode (HFM), can be read from the `MSR_PLATFORM_INFO` MSR_{0xCE}. The LFM is reflected in MSR_{0xCE}[47:40], and HFM in MSR_{0xCE}[15:8]. The former, the maximum frequency that can be supported by the minimum processor voltage, is also known as the maximum efficiency ratio. The latter is sometimes called the TSC frequency, because the Time Stamp Counter (TSC), a MSR that increments every clock cycle, always counts at this HFM frequency.

On systems with Turbo Boost technology, the highest guaranteed P-state that software can request for is the P1 state, which corresponds to the HFM. When Turbo Boost is enabled, (this can be done by zero-ing the bit at MSR_{0x1A0}[38]), only then can the highest P-state, P0, be

requested by software. The PCU is activated into Turbo mode when software requests for P0. Unlike the other P-states, P0 represents an opportunistic range of performance states that the PCU can transition the system into, but with a range of maximum Turbo frequency limits. These limits vary depending on the number of active cores, and can be read from and written to the MSR_TURBO_RATIO_LIMIT MSR_{0x1AD}. Every 8 bits in this MSR represents the maximum Turbo mode clock ratio limit for different number of active cores.

Not only can software read the current Turbo ratio limits, it can also re-configure these limits provided the processor allows them to be modified. A bit value of 1 in $MSR_{0xCE}[28]$ indicates that the limits can be changed. The bash script in Listing 6.1 demonstrates how these frequency limits for both non-Turbo and Turbo modes can be read from the MSRs. Using this bash script, we can read the non-Turbo LFM and HFM frequencies, as well as the Turbo maximum frequencies.

Listing 6.1: Bash script to read key frequency limits

```
#!/bin/bash
# Filename: read_turbo.sh

BCLK=0.1

lfm_ratio=`rdmsr -df 47:40 0xce`
lfm=`echo "$BCLK*$lfm_ratio" | bc -l`
hfm_ratio=`rdmsr -df 15:8 0xce`
hfm=`echo "$BCLK*$hfm_ratio" | bc -l`

t_on=`rdmsr -f 38:38 0x1a0`
if ((t_on==0));then t_on=ON;else t_on=OFF; fi

tc_on=`rdmsr -f 28:28 0xce`
if ((tc_on==1));then tc_on=YES;else tc_on=NO; fi

t_ratio1=`rdmsr -df 7:0 0x1ad`
t_ratio2=`rdmsr -df 15:8 0x1ad`
t_ratio3=`rdmsr -df 23:16 0x1ad`
t_ratio4=`rdmsr -df 31:24 0x1ad`
t_fq1=`echo "$BCLK*$t_ratio1" | bc -l`
t_fq2=`echo "$BCLK*$t_ratio2" | bc -l`
t_fq3=`echo "$BCLK*$t_ratio3" | bc -l`
t_fq4=`echo "$BCLK*$t_ratio4" | bc -l`

printf "Non-Turbo LFM: %.1f GHz\n" ${lfm}
printf "          HFM: %.1f GHz\n" ${hfm}
printf "Turbo mode: %s\n" "${t_on}"
printf "- 1 active core: %.1f GHz\n" ${t_fq1}
printf "- 2 active core: %.1f GHz\n" ${t_fq2}
printf "- 3 active core: %.1f GHz\n" ${t_fq3}
printf "- 4 active core: %.1f GHz\n" ${t_fq4}
printf "Programmable Turbo ratio limits: %s\n" "${tc_on}"
```

```
# ./read_turbo.sh
Non-Turbo LFM: 0.8 GHz
          HFM: 4.0 GHz
Turbo mode: ON
- 1 active core: 4.4 GHz
- 2 active core: 4.4 GHz
- 3 active core: 4.3 GHz
- 4 active core: 4.2 GHz
Programmable Turbo ratio limits: YES
```

The Turbo mode is controlled by the PCU hardware in most Intel processors, so the maximum P-state mode with software writes to MSR_{0x199} is the HFM. Any values above the HFM written to MSR_{0x199} will be ignored. However, to cater to users who are enthusiastic about overclocking their systems, Intel has released a series of processors with unlocked core clock multipliers that can be made to exceed the HFM ratio. These are known as the K-series or X-series processors. On these processors, software can set $Ratio_{CORE}$ to values above the HFM, albeit subject to the maximum Turbo ratio limits.

While these maximum Turbo ratio limits can be changed, there are hard ceiling to how much these Turbo ratio limits can be raised. These hard limits are typically fused at the hardware level and cannot be changed by software. The process of raising these Turbo limits beyond the default is commonly referred to as *overclocking* (OC). In the later Section § 6.1, we shall see how to read these fused Turbo ratio hard limits on the newer Haswell processor.

A.2.2.3 Empirical Measurements of Core V/F

Now that we have described the EIST and Turbo Boost technologies, it is helpful to observe them live in action on a real system. Since the operating voltage are changed in tandem with the core frequencies, we want to measure the different combinations of both frequencies and voltages of the cores at each P-state.

The core frequency can be read using the method described in the preceding sections. As for the measurement of voltage, we take advantage of a new field introduced into the `IA32_PERF_STATUS` MSR_{0x198}[47:32] in the Haswell processors. The voltage regulators (VR) in the processors supply the requested voltage based on a number of digital signal bits called Voltage Identification Digital (VID) lines. The 13-bit VID codes used by Intel VRs can be read from this MSR field, which gives a coded Voltage ID (VID) number we can then use to derive the currently supplied core voltage. To get the current core voltage in V, we divide the VID by 2^{13}. Listing 6.2 describes the bash script that can be used to read both the instantaneous core frequency and voltage.

Listing 6.2: Bash script to read core frequency and voltage

```
#!/bin/bash
# Filename: read_vf.sh

BCLK=0.1

freq_mul=`rdmsr -df 15:8 0x198`
volt_vid=`rdmsr -df 47:32 0x198`
freq=`echo "$BCLK*$freq_mul" | bc -l`
volt=`echo "$volt_vid/2^13" | bc -l`

printf "freq: %.1f GHz\n" "${freq}"
printf "volt: %.3f V\n" "${volt}"
```

```
# ./read_vf.sh
freq: 1.5 GHz
volt: 0.832 V
```

We run a variant of above measurement script that continuously records the combination of frequency and voltages on the system over a period of time. We use the default *ondemand*

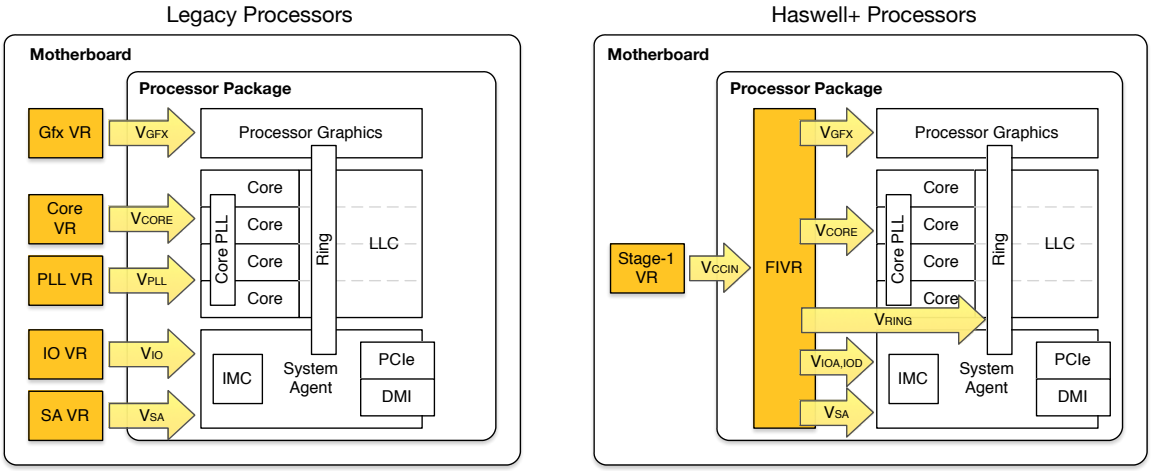

Figure 6.1.3: Comparison of voltage regulator (VR) design in legacy vs Haswell+ processors.

CPUfreq governor that dynamically requests for P-state changes based on the system utilization. The system is exercised to perform a variety of different workloads.

Figure 6.1.2 summarizes the empirical measurements of the various discrete combinations of frequency and voltages across different P-states. Each red cross on the diagram shows a specific frequency/voltage combination. While the frequency is changed by software in the non-Turbo mode, we see how the supply voltage is adjusted by the hardware PCU in real time. Note that each P-state from P1 to Pn is discrete state in terms of the frequencies. The P0 state is in fact a range of states with different frequencies managed by the PCU in the Turbo mode. This figure neatly demonstrates the interplay of the two key technologies that enable the seamless dynamic scaling of core frequency and voltages at runtime.

A.2.3 Recent PM Advances in Haswell

A.2.3.1 Fully Integrated Voltage Regulator

In the pre-Haswell processors, multiple off-die voltage regulators deliver the required power to various components in the processor package. To address the need for more individually controlled voltage rails and the need for a smaller physical footprint for newer mobile form factors, starting from the Haswell processors, Intel combines the various disparate voltage regulators into a Fully Integrated Voltage Regulator (FIVR) onto the processor package. Apart from the obvious benefit of area savings, the FIVR gives rise to the following optimization opportunities:

- Lower transition latencies when changing voltages for different domains
- More available power to accommodate burst performance, where the FIVR has at its disposal the entire package power to channel power to the domain that needs the most power at runtime
- A one-stop solution for regulating voltage changes for different domains

A.2.3.2 Extension of Clock/Voltage Domains

The Haswell processors enable finer-grained control over the clock frequencies by separating the cores and LLC/ring into separate clock domains. Unlike in the older processors where the cores and the LLC/ring have to operate at the same frequency, separating the different components into separate clock domains allows scope for optimizing the individual operating frequencies for enery-efficient performance according to the nature of the workload. For example, for memory-bound workloads, we can capitalize on the fact that CPU is not fully utilized while waiting for the memory units to complete the operation. We can thus achieve energy savings by having the option of decreasing the core frequency (and voltage) separately without having to decrease the LLC/ring frequency.

Major Architectural Domains. In Figure 6.1.1, we see how the base clock signal is distributed to various components in the processor. Several components can share the same clock signal provided by a PLL source, in which case we refer to them as sharing a single clock domain. Like a clock domain, a voltage domain encompasses a group of components sharing the same voltage source controlled by a VR. Both clock and voltage domains constitute the following major independent architectural domains found in Haswell processors:

- **Core**: The cores in the processor packages
- **Graphics (GFX)**: Integrated graphics processor
- **Ring**: High bandwidth communication interconnect designed in a ring topology to connect the cores, graphics processor, system agent and lower-level cache
- **System Agent (SA)**: The uncore portion of the processor that includes the DRAM integrated memory controller, display engine and integrates the Direct Media Interface (DMI) and PCI Express (PCIe) controller
- **I/O Analog (IOA)**: (This domain only applies to voltage)
- **I/O Digital (IOD)**: (This domain only applies to voltage)

A.2.4 New PM Controls in Haswell

With several key advances in power management in the Haswell processors, a number of new PM "knobs" and controls are introduced to provide additional ways for software to interface with the underlying PCU and FIVR. In this section, we describe these new controls and ways to use them for dynamic power management.

A.2.4.1 Newly Introduced MSR for PM

Intel introduces a new MSR in the Haswell processors to enable *runtime* fine-grained control of various voltage regulation and miscellaneous functionality related to overclocking. This MSR has an index of `0x150` and is readable and writable from software with kernel privileges. For lack of a name in official documents and the fact that there are some oblique references to this MSR as the *mailbox* MSR, we term this new MSR `MSR_OC_MAILBOX`. We show the various fields and the corresponding bit offsets in this MSR in Figure 6.1.4.

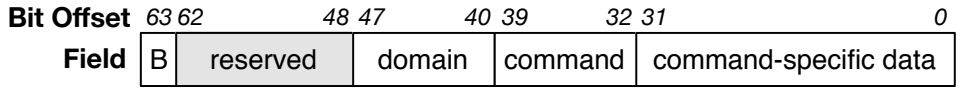

Figure 6.1.4: Newly introduced `MSR_OC_MAILBOX`.

At the instruction level, as with other MSRs, software interfaces with this `MSR_OC_MAILBOX` MSR with the x86 `rdmsr` and `wrmsr` instructions. However, its implementation is slightly different from the rest of the MSRs. Both configuring and interrogating settings with the `MSR_OC_MAILBOX` require two steps. First we need to issue a `wrmsr` to the MSR with a read or write command in one of the fields. Then we perform a `rdmsr` on the MSR to retrieve the configuration settings requested and a possible error code if the operation fails.

We will describe these operations in detail later in the section. We begin by giving an overview of the fields in `MSR_OC_MAILBOX`:

- **Command**: There are two types of commands, namely read and write commands, which we can use with the `MSR_OC_MAILBOX`. Listed under the *Command ID* column in Table 6.1.1, the read and write commands apply to different type of configuration settings. Some configuration are read-only, in which case the use of the write command will be ignored.
- **Command-specific data**: For read commands, the command-specific data field will contain the configuration settings returned from the MSR. For write commands, this field has to be populated with the desired settings for the MSR to propagate to the underlying hardware. This command-specific data consists of many sub-fields. We see a summary of these sub-fields under the *command-specific data* in Table 6.1.1.
- **Busy bit (B)**: Due to the stateful operation of the `MSR_OC_MAILBOX`, a special ready/busy bit is reserved to tell if a read/write operation is underway. A busy bit value of 1 indicates that this MSR is busy, in which case further reads or writes to this MSR are not allowed.
- **Domain**: Notice in Table 6.1.1, some commands are generic and some apply only to a specific domain. For the former commands, this domain field is set to 0. Conversely, for the latter, this field is used to indicate the architectural domain in a read/write request to this MSR. Each domain has a specific ID associated with it as follows – (Core: 0), (GFX: 1), (Ring: 2), (SA: 3), (IOA: 4), (IOD: 5).

Command ID		Command-Specific Data		
Read	**Write**	**Bit Offset**	**Sub-Field (Permissions)[1]**	**Description**
			GENERAL	
2	N.A.	[7:0]	Max $Ratio_{CORE}$ hard limit - 1C (RO)	Physically fused upper limits for Turbo ratio limits for $Ratio_{CORE}$.
		[15:8]	Max $Ratio_{CORE}$ hard limit - 2C (RO)	
		[23:16]	Max $Ratio_{CORE}$ hard limit - 3C (RO)	
		[31:24]	Max $Ratio_{CORE}$ hard limit - 4C (RO)	
18	19	[11:0]	SVID target voltage (RW)	-
		[31]	Disable SVID (RW)	-
20	21	[0]	Disable FIVR fault protection (RW)	Set 1 to disable protection circuits against excessive current and voltage.
		[1]	Disable FIVR efficiency management (RW)	-
			DOMAIN-SPECIFIC	
1	N.A.	[7:0]	Max clock ratio hard limit (RO)	Physically fused upper limit for clock ratio in this domain.
		[19:8]	Support for clock ratio overclocking (RO)	Bit value of 1 indicates support.
		[20]	Support for voltage static mode (RO)	
		[31:21]	Support for voltage offset (RO)*	
16	17	[7:0]	Max clock ratio (RW)	Refer to Section § 6.1 on using this for overclocking.
		[19:8]	Target static voltage (RW)	12-bit SVID to indicate target voltage value. Refer to Section § 6.1 for details on how to configure this.
		[20]	Voltage mode (RW)	Set to 1 for static mode, and 0 for default adaptive mode.
		[31:21]	Voltage offset (RW)*	-

[1] These sub-fields have different permissions. We denote Read-Only permission as RO and Read-and-Write permission as RW.

Table 6.1.1: Configuration fields in the `MSR_OC_MAILBOX`. (* - *Only these two fields are valid for use in the SA, IOA and IOD domains.*)

A.2.4.2 Basic MSR_OC_MAILBOX Operations

We describe the key steps in initiating read or write operations on MSR_OC_MAILBOX using the python-style code snippet in code Listing 6.3. Depending on what read or write commands used, the sub-fields in the command-specific data are populated with specific format. Table 6.1.1 describes these command-specific sub-fields in detail. Care must be taken to ensure that the busy bit is not set when reading from or writing to the MSR. This is a snippet from the full Python code Listing 6.3, which contains the various helper functions and scaffolding code.

Listing 6.3: Python-style code to read or write to OC MSR

```
def rd_wr_ocmsr(command, domain, readop=True, data=None):

  # This method returns when MSR is ready for read/write
  wait_ocmsr_ready()

  # Build 'packet' value to initiate operation with MSR
  val = 0L
  val |= make(val, BUSY_BIT, 63, 1) # busy bit
  val |= make(val, domain, 40, 8)   # domain
  val |= make(val, command, 32, 8)  # read/write cmd

  # Write operations require additional data
  if not readop:
    val |= data                     # cmd-specific data

  # Two-step process to interrogate MSR
  wrmsr(MSR_OC_MAILBOX, val)
  wait_ocmsr_ready()
  val = rdmsr(MSR_OC_MAILBOX)

  # Check for an error code
  errcode = bitfield(val, 32, 8)
  if errcode != 0:
    raise Exception('ERROR reading from MSR_OC_MAILBOX')

  # Return the value for read operations
  if readop:
    return val
```

To demonstrate interfacing with MSR_OC_MAILBOX, we create a Python script (in code Listing 6.3 to read all possible PM-related settings via this MSR. We show a subset of the script output here. It is interesting to see how much additional PM-related information we can get using this newly introduced MSR. For example, in the output below, we easily learn that the maximum possible Turbo ratio that can be overclocked (a setting that is typically fused at the hardware level) is 80.

```
# python read_mb.py
          <...snip...>
Domain: Core
- Max clock ratio hard limit: 80
- Support for clock ratio overclocking: 1
- Support for voltage static mode: 1
- Support for voltage offset: 1
          <...snip...>
- Voltage mode: 0
- Target static voltage: 0
```

```
- Voltage offset: 0
          <...snip...>
Max core ratio hard limit:
- 1 active core : 80
- 2 active cores: 80
          <...snip...>
```

A.2.4.3 Fine-grained Voltage Management

In the earlier sections, we show empirically how voltage and frequency are changed in tandem at runtime. This is the default mode where the PCU manages the voltage settings dynamically as frequency changes are initiated by software. Using the newly introduced `MSR_OC_MAILBOX`, software has additional control over the management of voltage settings. Furthermore, control over the supply voltage can be managed at a per-domain basis.

Re-configuring the voltage settings on the core requires writing to the `MSR_OC_MAILBOX` with command ID of 17 and domain ID of 0. To configure the core voltage with a fix target value, we write a 12-bit SVID value to MSR_{0x150}[19:8] and set 1 to the bit MSR_{0x150}[20]. Not all SVID values correspond to a discrete voltage. In fact, several SVID values may map to the same voltage value. If we denote the SVID value as ID_V and the target voltage value as V, V can be computed as follows:

$$\text{target voltage, } V = \frac{\lfloor \frac{ID_V+1}{5} \rfloor \times 40 + 10}{2^{13}} \tag{6.1}$$

Put simply, every block of 5 consecutive SVID values results in a different target voltage. Changing the SVID value by 5 makes approximately 4.88mV adjustments to the target voltage. Using the equation 6.1, we will set the target core voltage to roughly 1.368V using a SVID value of `0x577`. Remember to read back the value of the MSR after write to verify that the return code is 0 (which indicates a successful operation). We shall assume the core frequency has been fixed at 1 GHz.

```
# wrmsr 0x150 0x8000001100157700
# rdmsr 0x150
0
# ./read_vf.sh
freq: 1.0 GHz
volt: 1.368 V
```

Other than setting the target voltage to a static value, we can only apply a positive or negative voltage offset to the currently configured voltage. This voltage offset can be applied to both the static and adaptive voltage modes. This voltage offset is specified as a 11-bit ID that works the same way as the SVID value – changing the offset ID by 5 changes the target voltage by roughly 4.88mV. Negative offsets are represented using the two's complement.

Continuing from the above example, we will proceed to apply a positive voltage offset of roughly 15mV. This is configured using an offset ID of +15.

```
# wrmsr 0x150 0x8000001101f57700
# rdmsr 0x150
0
```

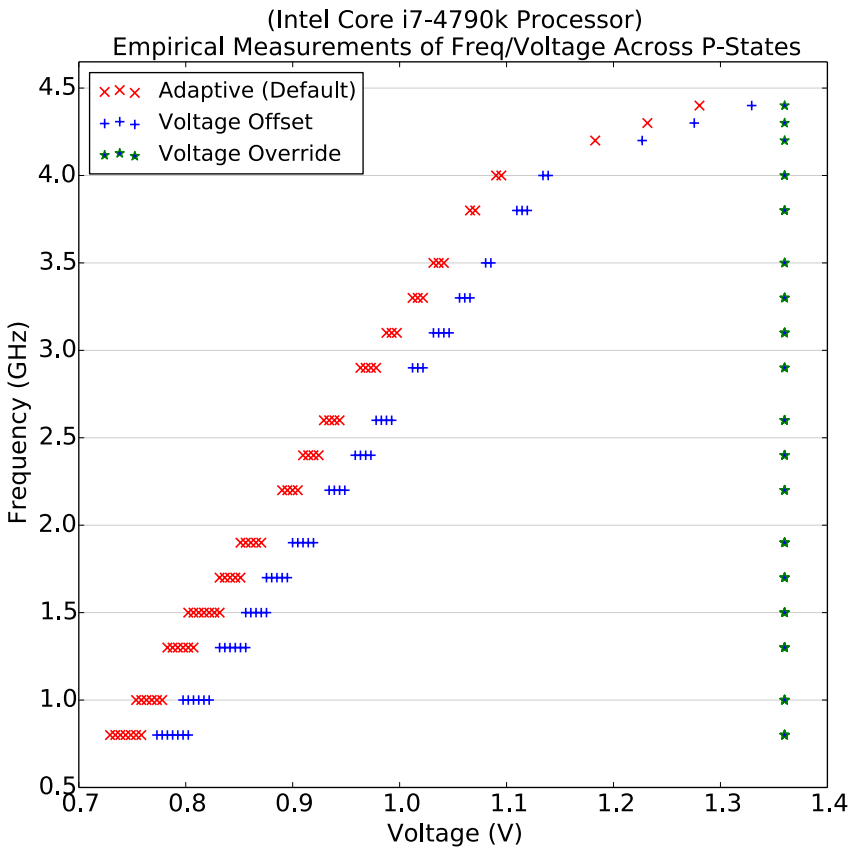

Figure 6.1.5: Freq/Voltage in different voltage modes.

```
# ./read_vf.sh
freq: 1.0 GHz
volt: 1.383 V
```

To reset the core voltage mode to be back to the adaptive mode (*i.e.* the PCU will adjust an appropriate voltage given the core frequency) and zero voltage offset, we set the voltage mode bit $MSR_{0x150}[20]$ to 0, together with the zero command-specific field.

```
# wrmsr 0x150 0x8000001100000000
# rdmsr 0x150
0
# ./read_vf.sh
freq: 1.0 GHz
volt: 0.773 V
```

To demonstrate that the core voltage settings can be configured via `MSR_OC_MAILBOX`, we use a variant of our measurement script (in code Listing 6.2) to record the voltage and frequency combinations, while we make changes to the core voltage settings.

In one instance, we change the core *Target static voltage* to 1.368V. In another instance, we apply an additional core *Voltage offset* of 48.8mV. Figure 6.1.5 shows the different combinations of measured core voltage and frequency as we make the various changes to the core voltage settings.

A.2.4.4 Overclocking with Turbo Ratio Limits

All Intel processors are shipped with hard limits to the core Turbo clock ratios. These limits are physically fused in hardware and cannot be changed in software. We have seen in Section § 6.1 how to read these limits from hardware. Running the script in Listing 6.3 will give us this limit, `Max clock ratio hard limit`, for the core domain.

Unlike the non-K series processors, K-series processors offer software the opportunity for further overclocking of the default Turbo ratio limits up to these hard limits[6]. We can see there is room for further increase of the various Turbo ratio limits.

```
# python read_mb.py
          <...snip...>
Max core ratio hard limit:
- 1 active core : 80
- 2 active cores: 80
          <...snip...>
# rdmsr 0x1ad
2a2b2c2c
```

Merely raising the Turbo ratio limits in the `MSR_TURBO_RATIO_LIMIT` MSR_{0x1AD} is insufficient to enable the overclocking. As we shall see below, even when we change the Turbo ratio limits to a maximum frequency of 4.5GHz using a clock ratio of 45 (hex value `0x2d`), the operating frequency can only be set to a maximum of the default Turbo ratio limits, *i.e.* 4.4Ghz.

```
# wrmsr 0x1ad 0x2d2d2d2d
# rdmsr 0x1ad
2d2d2d2d
# wrmsr 0x199 0x2d00
# ./read_vf.sh
freq: 4.4 GHz
volt: 1.280 V
```

Several steps are necessary to effectively perform overclocking with the Turbo ratio limits. In the physical memory of the processor, there is a range of memory-mapped I/O configuration space, MCHBAR, that comprise a range of registers related to functions including memory control, power and thermal control. To perform realtime overclocking, we need to access a register in this configuration space to reflect the raised Turbo ratio limit.

The physical address of this overclocking register is at the address offset `0x5990` of the MCHBAR base address. We can get the MCHBAR base address at offset `0x48` at physical memory of device `0` in the PCI configuration space. We will use `lspci` to first get the base address of MCHBAR, and then use `devmem2` to access the overclocking register.

```
# lspci -s00:00.0
00:00.0 Host bridge: Intel Corporation 4th [...]

# lspci -s00:00.0 -xxx | grep 40:
```

[6]These hard limits are only theoretical limits physically fused into hardware, and do not necessarily guarantee stable operation of the system for frequencies beyond the default Turbo ratio limits.

```
40: 01 90 d1 fe 00 00 00 00 01 00 d1 fe 00 00 00 00

# devmem2 0xfed15990
/dev/mem opened.
Memory mapped at address 0x7f04b97f9000.
Value at address 0xFED15990 (0x7f04b97f9990): 0xFF
```

We now know that the physical address of the overclocking register is `0xFED15990`. Raising the Turbo ratio limits consists of the following steps:

1. Get base physical address of MCHBAR (*i.e.* `0xfed10000`), and the physical address of overclocking register (*i.e.* `0xfed15990`)
2. Raise Turbo ratio limits in MSR_{0x1AD}
3. Write the maximum Turbo ratio limit to the overclocking register
4. Update the target operating core ratio in `IA32_PERF_CTL` MSR_{0x199}[15:8]
5. Issue write command for the core domain with the target operating core ratio in `MSR_OC_MAILBOX` MSR_{0x150}[7:0]

We demonstrate the concrete steps using the bash shell as follows:

```
# ./read_vf.sh
freq: 1.0 GHz
volt: 0.778 V

# rdmsr 0x1ad
2a2b2c2c

# wrmsr 0x1ad 0x2d2d2d2d

# rdmsr 0x1ad
2d2d2d2d

# devmem2 0xfed15990 b 0x2d
/dev/mem opened.
Memory mapped at address 0x7fc71a6ca000.
Value at address 0xFED15990 (0x7fc71a6ca990): 0x2C
Written 0x2D; readback 0x2D

# wrmsr 0x199 0x2d00

# wrmsr 0x150 0x800000110000002e

# ./read_vf.sh
freq: 4.5 GHz
volt: 1.280 V
```

MSR Name / Index	Compat.[1]	Bit Offset	Field (Permissions)	Description
MSR_PLATFORM_INFO 0xCE	NHM - *	[15:8]	Max non-Turbo ratio (RO)	Maximum non-Turbo P-state - HFM.
		[28]	Adjustable Turbo ratio limits (RO)	Indicate if the Turbo ratio limits can be changed.
		[29]	Power limit enable (RO)	-
	IVB - *	[34:33]	Number of config TDP levels (RO)	-
		[47:40]	Maximum efficiency ratio (RO)	Minimum P-state - LFM.
MSR_OC_MAILBOX 0x150	HSW - *	[31:0]	Command-specific data (RW)	Newly introduced MSR to interface with PCU and FIVR. Refer to Section § 6.1.
		[39:32]	Command (RW)	
		[47:40]	Domain (RW)	
		[63]	Busy bit (RW)	
MSR_FLEX_RATIO 0x194	SNB - IVB	[7:0]	OC extra voltage (RW)	-
	SNB - *	[19:17]	Number of OC bins supported (RO)	-
	IVB - *	[20]	Overclocking (OC) lock (RWL)[2]	Set to 1 to disable OC.
IA32_PERF_STATUS 0x198	SNB - *	[15:8]	Current P-state ratio (RO)	Measured $Ratio_{CORE}$.
	HSW - *	[47:32]	Current P-state voltage VID (RO)	Measured VID.
IA32_PERF_CTL 0x199	SNB - *	[15:8]	Target P-state ratio (RW)	Request P-state change by setting this to $Ratio_{CORE}$. Refer to Section § 6.1.
		[32]	Disable Turbo mode (RW)	Set to 1 to disable Turbo mode.
IA32_MISC_ENABLE 0x1A0	NHM - *	[16]	EIST mode enabled (RW)	Set 1 to enable EIST.
		[38]	Disable Turbo mode (RW)	Set to 1 to disable Turbo mode.
MSR_TURBO_RATIO_LIMIT 0x1AD	NHM - *	[7:0]	Max ratio limit for 1 core (RW)	Indicate max core clock ratio in Turbo mode for different number of active cores. Refer to Section § 6.1.
		[15:8]	Max ratio limit for 2 core (RW)	
		[23:16]	Max ratio limit for 3 core (RW)	
		[31:24]	Max ratio limit for 4 core (RW)	

[1] The fields in the MSRs may only apply to specific architecture. We denote the major Intel architecture as follows: (NHM: Nehalem), (SNB: Sandy Bridge), (IVB: Ivy Bridge), (HSW: Haswell). For example, "NHM - *" means that this sub-field can be found in the processors of Nehalem or later architecture.

[2] This sub-field has RW permissions. Hardware can make this sub-field Read-Only via a separate configuration bit or logic.

Table 6.1.2: Summary of PM-related Intel MSRs and corresponding fields.

A.2.5 Intel PM MSRs in a Nutshell

To provide a quick reference for readers, we summarize all the Intel MSRs related to power management and their corresponding fields in Table 6.1.2.

A.3 Code Availability

The code for the respective projects are available at the following repositories:

- `CLKscrew`: `https://github.com/0x0atang/clkscrew`
- `Heisenbyte`: `https://github.com/0x0atang/heisenbyte`
- `Hades`: `https://github.com/0x0atang/hades`