Leveraging Processor Features for System Security

by

Zelalem Birhanu Aweke

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2019

Doctoral Committee:

Professor Todd M. Austin, Chair
Assistant Professor Reetuparna Das
Assistant Professor Jean-Baptiste Jeannin
Assistant Professor Baris Can Cengiz Kasikci

Zelalem Birhanu Aweke

zaweke@umich.edu

ORCID iD: 0000-0002-6899-6938

© Zelalem Birhanu Aweke 2019

To my family

Acknowledgments

First and foremost I would like to thank my adviser, Professor Todd Austin, for his constant support and for being a great mentor. Todd has taught me a great deal about how to be a good researcher and how to effectively communicate my work.

I would also like to thank my thesis committee members, Professor Reetuparna Das, Professor Baris Can Cengiz Kasikc and Professor Jean-Baptiste Jeannin for their support and valuable feedback.

I am especially fortunate to have worked with Professor Reetuparna Das at the beginning of my graduate school career. She had been patient with me and taught me a lot about research while I transition from being a mere student to a researcher. I Would also like to thank my collaborators, Salessawi Ferede Yitbarek and Matthew Hicks, who played major roles in my first research project.

I would like to thank my friends and colleagues at the University of Michigan. Salessawi Ferede Yitbarek, Biruk Mammo, William Arthur and Patipan Prasertsom were immensely helpful while I settle in Ann Arbor. I would also like to thank Misiker Aga, Abraham Addisie, Doowon Lee, Pete Ehrett, Brendan West, Lauren Biernacki, Mark Gallagher, Colton Holoday, Taru Verma, Vidushi Goyal, Timothy Linscott, Andrew McCrabb, Hiwot Kassa, Leul Belayneh, Nishil Talati, Kidus Admassu, Abeselom Fanta and a lot more others that I havent mention here - who helped me navigate through grad school life, be it in discussing ideas, reviewing my papers or just being friends.

My stay at the University of Michigan has gone smoothly, thanks to the wonderful CSE staff. I would especially like to thank Ashley Andreae, Dawn Freysinger, Christine Boltz, Alice Melloni, Tracie Straub, Laura Pasek, Denise DuPrie, Erika Hauff, Karen Liska and Stephen Reger for their great assistance.

Finally, I would like to thank my family who has been extremely supportive and continue to support me in my endeavors, especially my parents who, despite receiving little education themselves, understood its value and did everything they could to make sure I focus on what is important to me. I am also blessed with great brothers and sisters who have always been there for me. Thank you all.

Table of Contents

Dedication ii

Acknowledgments iii

List of Figures vii

List of Tables viii

List of Algorithms ix

Abstract x

Chapter 1 Introduction 1
- 1.1 Memory Corruption Vulnerabilities 1
- 1.2 Memory Corruption Exploit Mitigations 2
- 1.3 Contributions of the Thesis 5
  - 1.3.1 Efficient Temporal Memory Safety with Pointer Authentication . . 5
  - 1.3.2 Efficient Software-based Fault Isolation for Embedded Devices . . 7
  - 1.3.3 Detecting Rowhammer Attacks with Hardware Performance Counters 8

Chapter 2 Pointer Authentication for Efficient Temporal Memory Safety . . . 13
- 2.1 Introduction 13
- 2.2 Background 14
  - 2.2.1 The Lock-and-key Technique 14
  - 2.2.2 ARM Pointer Authentication 15
  - 2.2.3 The Low-fat Memory Layout 16
- 2.3 Pointer Authentication for Temporal Safety 18
  - 2.3.1 PETS with Shadow Storage (Baseline PETS) 19
  - 2.3.2 Low-fat PETS 21
- 2.4 PETS Implementation 22
  - 2.4.1 PETS Compiler Instrumentation 22
  - 2.4.2 Memory Allocators 24
  - 2.4.3 Support for Multithreaded Applications 25
- 2.5 Experimental Evaluation 26

2.5.1 Experimental Setup 26
2.5.2 Runtime Overhead: 29
2.5.3 Memory Overhead 29
2.5.4 Comparison with other Temporal Safety Techniques 30
2.6 Related Work 32
2.6.1 Temporal Safety Techniques 34
2.7 Chapter Summary 36

Chapter 3 Efficient Software Fault Isolation for IoT-Class Devices 37
3.1 Introduction 37
3.2 Threat Model 39
3.3 **u**SFI System Architecture 39
3.3.1 Module Privilege Levels 40
3.3.2 Memory Isolation 42
3.3.3 uSFI Compiler and Verifier 43
3.3.4 uSFI Runtime 48
3.4 **u**SFI Implementation 51
3.4.1 MPU Configuration 51
3.4.2 uSFI Compiler and Verifier 52
3.4.3 uSFI Runtime 53
3.5 Experimental Evaluation 54
3.5.1 Code Size Overhead 55
3.5.2 Performance Overhead 56
3.5.3 Case Study 1: Step Analysis 57
3.5.4 Case Study 2: HTTPS File Download 58
3.5.5 Trusted Code Size 59
3.6 Related Work 60
3.6.1 Software-based Fault Isolation 60
3.6.2 Sandboxing in Embedded Devices 60
3.7 Chapter Summary 63

Chapter 4 Detecting Rowhammer Attacks with Hardware Performance Counters 64
4.1 Introduction 64
4.2 Breaking Current Mitigation Techniques 65
4.2.1 Rowhammering under a Double Refresh Rate 66
4.2.2 Rowhammering without the CLFLUSH instruction 67
4.3 Software-Based Rowhammer Detection and Protection 72
4.3.1 Detecting Rowhammer Attacks 72
4.3.2 Protecting Potential Rowhammer Victims 73
4.3.3 ANVIL: A Linux-Based Rowhammer Protection Mechanism 74
4.4 Experimental Evaluation 78
4.4.1 Benchmark Applications 78
4.4.2 Rowhammer Detection Characteristics 78

4.4.3 Performance Evaluation . . . 80
4.4.4 Comparison with Increased Refresh Rate . . . 81
4.4.5 Robustness to Potential Future Rowhammer Attacks . . . 82
4.5 Related Work . . . 83
4.5.1 Rowhammer Vulnerability and Its Exploitation . . . 84
4.5.2 Rowhammer Mitigations . . . 86
4.6 Chapter Summary . . . 88

Chapter 5 Conclusion . . . 90
5.1 Future Directions . . . 90

Bibliography . . . 93

List of Figures

1.1 Root Causes of Critical Vulnerabilities for the Past Five Years 2
1.2 Control-flow Hijacking Attack and Mitigations 3

2.1 The Lock-and-Key Technique 15
2.2 ARM Pointer Authentication – Computing the PAC 16
2.3 Low-fat Memory Layout 17
2.4 Metadata Storage and Access 18
2.5 Percentage of Redundant Memory Access Checks Removed by PETS Compiler Optimization 28
2.6 PETS Runtime Overhead 28
2.7 PETS Memory Overhead 30

3.1 uSFI System Components 40
3.2 uSFI Compiler and Verifier 44
3.3 Potential Illegal Instructions and How to Fix Them 45
3.4 PC-relative Load Instruction and its Conversion to Safe Move Instructions . 46
3.5 Inter-module Function Call 49
3.6 uSFI Interrupt Handling 50
3.7 uSFI Compiler Implementation 52
3.8 ARM Wearable Reference Design Step Analysis Application 57
3.9 Software Modules for HTTPS File Download Application 58

4.1 Memory Access Patterns for CLFLUSH-based & CLFLUSH-free Double-sided Rowhammer Attacks 68
4.2 Software-Based Rowhammer Attack Detector 74
4.3 ANVIL's Impact on Non-Malicious Programs 81
4.4 Sensitivity of Execution Overheads to Potential Future Attacks 84

List of Tables

2.1 Memory Overhead Comparison 32
2.2 Use-after-detection Mechanism and Metadata Management for Temporal Safety Techniques 33

3.1 uSFI Privilege Levels and Allowed Access 42
3.2 Access Permission Configuration for Module Memory Regions 43
3.3 Code Size Overhead 56
3.4 Execution Cycles for the Step Analysis Application 59

4.1 Rowhammer Attack Characteristics 66
4.2 Rowhammer Detector Parameters to Evaluate Accuracy of Rowhammer Detection 79
4.3 Rowhammer Detection Result for Rowhammering Programs 80
4.4 Rate of False Positive Refreshes 82
4.5 Rate of False Positive Refreshes for ANVIL-Heavy and ANVIL-Light . . . 84

List of Algorithms

2.1 Algorithm for UAF Detector Pass 23
3.2 Algorithm for uSFI Verifier 47

Abstract

Errors in hardware and software lead to vulnerabilities that can be exploited by attackers. Proposed exploit mitigation techniques can be broadly categorized into two: software-only techniques and techniques that propose specialized hardware extensions. Software-only techniques can be implemented on existing hardware, but typically suffer from impractically high overheads. On the other hand, specialized hardware extensions, while improving performance, in practice require a long time to be incorporated into production hardware. In this dissertation, we propose adapting existing processor features to provide novel and low-overhead security solutions.

In the first part of the dissertation, we show how modern hardware features can be used to provide efficient memory safety. One component of memory safety that has become important in recent years is temporal memory safety. Temporal memory safety techniques are used to detect memory errors such as use-after-free errors. This dissertation proposes a temporal memory safety technique that takes advantage of pointer authentication hardware to significantly reduce the memory and runtime overhead of traditional temporal safety techniques. Providing complete memory safety on resource constrained devices is expensive, therefore we propose software-based fault isolation (sandboxing) as an efficient alternative to constrain attackers' access to code and data in embedded systems. We show how we can use the memory protection unit (MPU) hardware available in many embedded devices along with a small trusted runtime to build a low-overhead sandboxing mechanism.

In the second part of the dissertation, we show how hardware performance counters in modern processors can be used to detect rowhammer attacks. Our technique detects rowhammer attacks by monitoring for high locality memory accesses out of the last-level cache using hardware performance counters. The technique accurately detects rowhammer attacks with a low performance overhead and without requiring hardware modifications.

Chapter 1
Introduction

In recent years it has become common to hear news stories such as ransomware attacks forcing hospitals to shutdown [1], power plants being at risk due to software flaws [2] and companies rolling out patches to stop major security bugs [3, 4]. Such stories typically begin with software bugs such as memory corruption bugs. These bugs create vulnerabilities that can be exploited by attackers to, for example, leak memory, escalate privilege or execute arbitrary code. One of the most exploited software vulnerabilities are memory corruption vulnerabilities.

1.1 Memory Corruption Vulnerabilities

Many programs including OS kernels and runtime environments are written in unmanaged languages such as C and C++. While programs written with these languages can achieve high performance, the fact that memory is managed by the programmer makes them prone to memory corruption errors such as buffer overflow and use-after-free errors. These errors have historically been one of the most exploited errors and still persist today. Figure 1.1 shows a study of the root causes of critical vulnerabilities for a five year period. The data is compiled from [5]. From the graph we can see that memory corruption errors (buffer overflow and use-after-free errors) are still the number one causes of critical vulnerabilities. In 2017 alone, they made up more than 50% of the vulnerabilities.

Memory corruption errors are typically used as a basis for exploits such as control-flow hijacking or data-oriented programming [6, 7]. Figure 1.2 shows how memory corruption

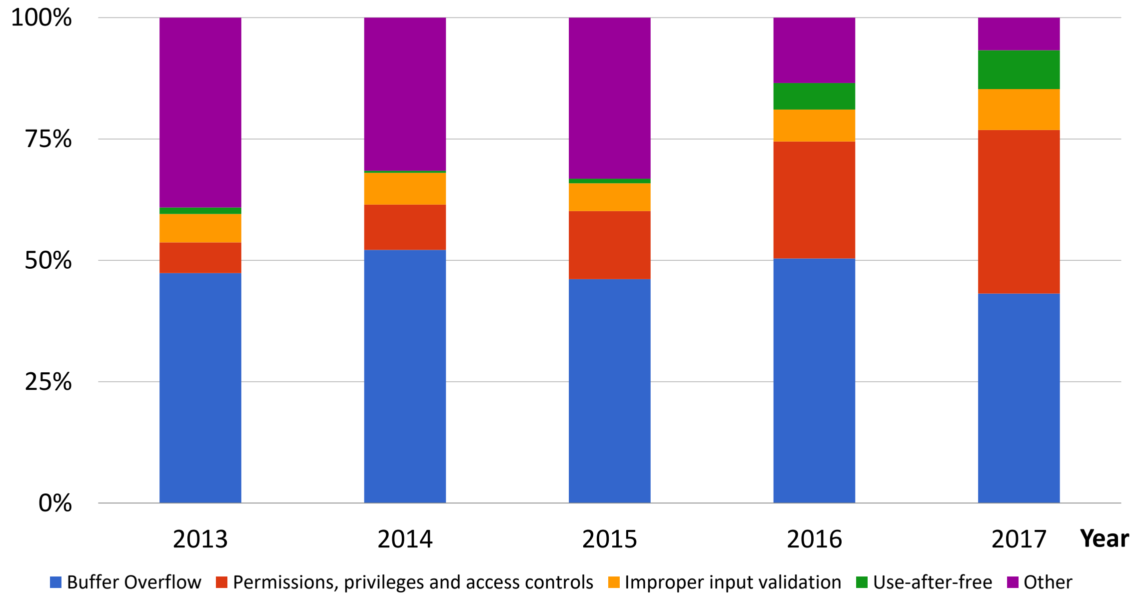


Figure 1.1 Root Causes of Critical Vulnerabilities for the Past Five Years.

errors are used to mount an attack, taking control-flow hijacking attack as an example [8]. The first step of a control-flow hijacking attack is making a pointer go out-of bounds using a buffer overflow vulnerability or accessing a dangling pointer by using a use-after-free error. Assuming now the pointer points to a code pointer, we can use it to modify the value of the code pointer so that the code pointer points to exploit code. Then using an indirect call, an indirect branch or a return instruction, control is transferred to the exploit code. The exploit code can be new code injected by the attacker or chained together using already existing code (*code gadgets*). The exploit code typically starts a command shell to allow the attacker to gain control of the system (i.e. to execute arbitrary code and access arbitrary data).

1.2 Memory Corruption Exploit Mitigations

Various defenses against memory corruption exploits have been proposed. These techniques target the various stages of the exploit as shown on Figure 1.2. A group of techniques, called *memory safety* techniques, target the root causes of the exploits - the memory corruption

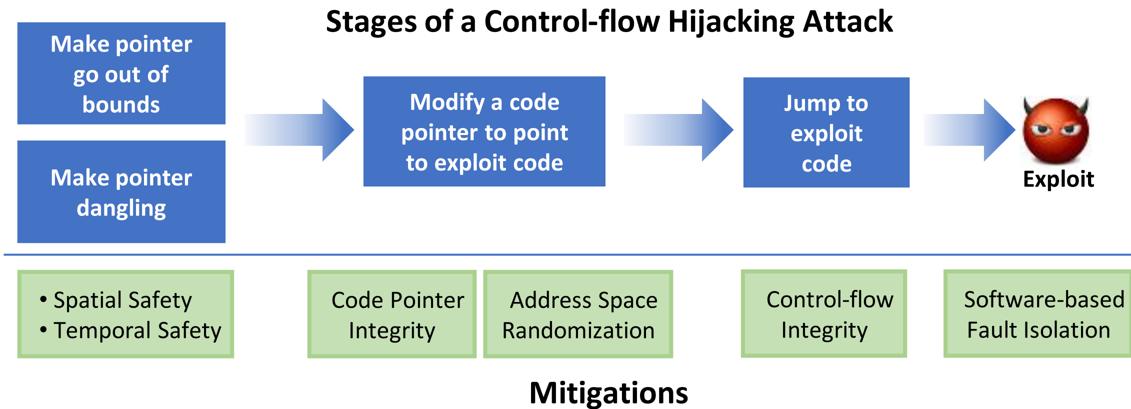


Figure 1.2 Control-flow Hijacking Attack and Mitigations. The figure shows the stages of mounting a control-flow hijacking attack and the different mitigation techniques that have been proposed. The mitigations target the different stages of the attack.

---

errors themselves. These techniques are categorized as *spatial safety* or *temporal safety* techniques. Spatial safety techniques detect spatial errors such as buffer overflows [8]. This typically involves transforming code to include bounds checking instructions. Temporal safety techniques detect temporal errors such as use-after-free errors using various approaches [9, 10, 11, 12, 13, 14].

Enforcing complete memory safety would stop all memory-corruption-based attacks; however, its performance overhead would be very high. For this reason, other techniques that target the later stages of the control-flow hijacking attack have been proposed. Assuming there is a potential memory corruption error, one thing that can be done is prevent the corruption of code pointers at the second stage on Figure 1.2. That is what *Code Pointer Integrity (CPI)* does [15]. Other techniques attempt to hide the addresses of payload code or gadgets by randomizing the locations of code and data. *Address Space Layout Randomization (ASLR)* [16] is an example of such a technique.

Assuming the attacker knows the location of the payload code or gadgets, and they can corrupt a code pointer, the next step is to jump to that code. This entails diverting the execution from the control-flow defined by the source code. *Control-flow Integrity (CFI)* solutions that enforce some policy regarding indirect control transfers can detect control-

flow hijacking attacks at this stage. If the attack makes it past this stage, the control-flow hijacking attack is considered a success and the attacker is assumed to have some control of the system. At this stage, higher-level defences such as *Software-based fault Isolation (sandboxing)* can be used to confine the attacker's access to code and data (*i.e.*, the attacker is only allowed to access code and data defined by an access control policy).

Typically, mitigations are implemented using software-only techniques. While software-only techniques can be implemented on existing hardware, they suffer from high performance overheads. For example, complete memory safety (*i.e.*, a technique that provides both spatial and temporal memory safety) has an average slow down of more than 2x [17]. To reduce the overhead of software-only techniques, specialized hardware extensions have been proposed (*e.g.* [18, 19, 20]). Hardware extensions can significantly reduce the overhead, however the techniques can not be used on existing hardware. In addition to this, in practice it takes a long time for a hardware extension to be incorporated into production hardware. For example, control-flow integrity (CFI) [21] is one of the most popular control flow security techniques. Even though hardware support for CFI has been proposed as early as 2005, it is only recently that Intel announced its plan to include hardware support for CFI in future processors [22].

A middle ground in addressing the issue of high overhead in software-only techniques and usability in specialized hardware extensions is adapting existing processor features. Modern processors are rich with hardware features. For example, the x86 architecture has introduced more than 20 ISA extensions since 2011 [23]. We can use these features along with software techniques to design more practical security solutions without requiring specialized hardware extensions.

1.3 Contributions of the Thesis

In this dissertation, we take the approach of adapting existing processor features to provide novel and practical security solutions. We present three works that demonstrate this approach. In Chapter 2, we present an efficient temporal memory safety technique that uses the pointer authentication feature in ARM processors. In Chapter 3, we present a technique that uses memory protection unit hardware to provide an efficient sandboxing capability in an embedded device. Finally, in Chapter 4, we show how we can detect rowhammer attacks using hardware performance counters.

1.3.1 Efficient Temporal Memory Safety with Pointer Authentication

As mentioned in the previous section, temporal safety is the first line of defense against use-after-free errors. Use-after-free errors occur when an object is accessed outside of the time during which it was allocated (after the object has been freed). As shown on Figure 1.1, they are among the top four critical vulnerabilities and have been consistently on the rise. For example, from the year 2015 to 2017, the number of critical use-after-free errors has increased by 8x. In the 2017 Pwn2Own hacking contest, more than half of the exploits used a use-after-free vulnerability [24].

The Lock-and-key Temporal Safety Technique: A number of temporal safety techniques have been proposed over the years [9, 10, 11, 12, 13, 14]. These techniques vary from those that provide probabilistic protections to those that guarantee temporal safety with varying degree of compatibility, complexity and performance overhead. One such technique is the *lock-and-key* technique [9, 14, 25]. This technique assigns a key for each pointer to a memory object, and a matching lock value to the object. On every access to the memory object, a check is made to see if the lock and key values match. Access to the memory object is allowed only if a pointer key matches that of the object lock. When a memory object is deallocated, the lock value is changed, so that further accesses using dangling pointers will

be detected.

An important parameter that affects the performance of a lock-and-key technique is *metadata management* (*i.e*, how lock and key metadata is stored and accessed). Two aspects of metadata management are important:

(1) Granularity of metadata (*i.e.*, pointer metadata vs. memory object metadata): Previous lock-and-key defenses keep a *key* and *lock* pointer metadata *for each* pointer. Typically, there are more pointers than memory objects in a program, therefore keeping per-pointer metadata incurs higher memory overhead than keeping per-memory-object metadata.

(2) How metadata is stored: Some techniques replace pointers with *fat pointers* [14, 25] to store metadata. Fat pointers are structures that contain pointer metadata in addition to the original pointer value. Fat pointers do not add much additional cache pressure compared to regular pointers, however, they require extensive instrumentation of a program [26]. Other techniques have proposed using disjoint metadata storage [9, 18]. One such approach is using a direct-mapped shadow, where metadata is stored at a fixed offset from a pointer. However, shadowing every pointer with this approach incurs a high memory overhead. To reduce the memory footprint, CETS [9] uses a two-level trie data structure to store per-pointer metadata. While this approach has better compatibility than the fat-pointer-based approaches, the multiple loads required to access metadata incur a high runtime overhead. Furthermore, the need to explicitly propagate metadata when a new pointer is derived from an existing pointer makes code instrumentation complex. For example, function prototypes need to be replaced with new prototypes that include metadata.

The lock-and-key technique is effective in detecting use-after-free errors, however previous proposals suffer from complex program instrumentation and high memory and runtime overheads that arise from the way metadata is stored and accessed. In Chapter 2 of this thesis, we propose PETS, a lock-and-key temporal safety technique that takes advantage

of the recently introduced pointer authentication feature in the ARM architecture [27] to provide efficient temporal safety.

Pointer authentication stores a cryptographic hash of the pointer value, called a *pointer authentication code (PAC)*, in the unused bits of pointers. With PETS, PACs are used as a replacement for keys. This way lock metadata is kept *per memory object* instead of per pointer, reducing the metadata storage overhead. Furthermore, the relatively small metadata means we can use a direct-mapped shadow to store lock pointer metadata resulting in faster metadata access. Finally, because the key metadata is embedded in the pointer itself, passing pointers as function arguments doesn't require changes to the standard calling convention.

1.3.2 Efficient Software-based Fault Isolation for Embedded Devices

Embedded devices are exposed to the same memory corruption vulnerabilities that are common in traditional computing systems [28, 29, 30]. To make matters worse, many embedded systems lack fundamental code and data memory protection mechanisms that are available in more powerful computing systems. For example, low-end embedded processors typically do not include a memory management unit (MMU) to reduce cost and in some cases to provide real-time execution time guarantees [31]. Therefore, low-end embedded systems typically operate in a single address space with tightly-coupled software modules (e.g., RTOS kernels, peripheral drivers, libraries, etc.) without any form of isolation between modules. A bug in one software module can compromise the security of the whole system.

Previous works have proposed mechanisms to protect code and data in embedded systems, with varying degrees of security assurances and resource requirements. TrustLite [32] proposed a hardware extension to provide isolation of trusted modules (Trustlets) from untrusted code including an untrusted OS. While TrustLite provides strong data isolation guarantees, it requires a hardware extension. Most importantly, it is hardly scalable as the required area of the hardware extension grows linearly with the number of protected modules (the area of the hardware extension matches that of the core for nine modules).

Other works have proposed variants of software-based fault isolation (SFI) as a mechanism to isolate tightly-coupled software modules that share the same address space. ARMor [33] uses SFI to sandbox non-critical code by performing binary rewriting to put checks before store operations identified as being potentially unsafe. At runtime, it uses a separate control stack to protect return addresses. Similarly, [34] and [35] use a separate stack to protect return addresses. Other indirect control flow instructions are validated by runtime checks. These software-based protection mechanisms are attractive as they don't require hardware changes. However, this reduced hardware requirement comes at the cost of reduced performance and/or security. In order to reduce the overhead of the runtime checks, the techniques only provide write protection. A malicious software can read and leak sensitive data. In addition to this, the additional memory guard instructions result in larger code sizes and higher performance overheads. As such, there is a need for a *low-cost* mechanism that provides *strong* (read, write, and execution) protection.

In Chapter 3, we present uSFI, a low-cost code and data isolation mechanism for resource constrained embedded devices. uSFI uses readily available hardware, memory protection unit (MPU), along with static software analysis to provide stronger security guarantees at a lower cost than previous efforts. MPU allows partitioning memory into regions and assigning attributes and access permissions to each region. The MPU hardware enforces the access permissions, therefore there is no need to instrument every memory access instruction, eliminating the performance overhead associated with memory access checks. Further, uSFI provides both read and write protection, guaranteeing stronger security than previous works.

1.3.3 Detecting Rowhammer Attacks with Hardware Performance Counters

Rowhammer attacks exploit an electrical cross-talk property within the dense interconnect of modern DRAMs (also known as DRAM disturbance errors). Kim, et al. [36] showed that by repeatedly accessing a DRAM row referred to as an *aggressor row*, bits in adjacent DRAM

rows (called *victim rows*) can be flipped. To flip bits, the aggressor row has to be accessed 100's of thousands of times within a DRAM refresh period (typically 64ms). In order to bypass the cache and quickly access the aggressor row, the authors used x86's *CLFLUSH* instruction. Subsequently, Seaborn and Dullien [37] demonstrated two attacks that use rowhammering. The first attack used rowhammering to escape from the Native Client sandbox by rowhammering a code segment and rewriting an already verified instruction. In their second attack, they were able to gain read/write access to page table entries, essentially gaining access to all physical memory. These attacks showed that DRAM disturbance errors not only cause unexpected program behaviors or failures, but also present major security risks.

A number of mitigation techniques for the rowhammer problem have been suggested for both legacy and future systems. One technique that is currently in use is doubling DRAM refresh rate (*i.e.* reducing the refresh period from 64ms to 32ms) [38, 39, 40]. By doing this it is believed that there will not be sufficient time to generate enough DRAM row activations. But as has been suggested previously [36] and as we will show in Chapter 4, 32ms is more than sufficient to generate enough DRAM row activations to produce bit flips. A second protection mechanism used against rowhammering-based attacks is limiting access to cache flushing instruction CLFLUSH [37]. CLFLUSH allows quick access to DRAM by flushing a specific cache line, and restricting this instruction makes rowhammering non-trivial. But as we show in Chapter 4 and as shown by subsequent works [41, 42], rowhammering attack can be implemented with ordinary load and store instructions without requiring a cache flushing instruction.

There is a mention of the existence of protections against rowhammer errors on more recent devices [43, 44]. The LPDDR4 specification and new DDR4 modules include a targeted row refresh (TRR) capability designed to thwart rowhammer attacks. The mechanism tracks the number of row activations within a fixed time window, and selectively refreshes rows adjacent to a too-frequently accessed DRAM row. However, recent works have shown

that DDR4 is also susceptible to rowhammer attacks [41, 45].

Finally, the architecture literature has seen a few rowhammer protection proposals [36, 46]. For example, one proposal utilizes probabilistic adjacent row activation (PARA) to refresh the neighboring rows of any DRAM row access, with low probability [36]. The idea behind this approach is that the many repeated DRAM row accesses required to hammer a victim DRAM row will result in an early refresh of the victim row with extremely high (cumulative) probability [36]. However, such solutions require the introduction of new hardware, therefore wouldn't protect existing systems.

In Chapter 4 of this dissertation, we show how we can detect rowhammer attacks on existing systems. We make the observation that rowhammer attacks require *high-locality* misses out of the last-level cache. Therefore, it is possible to detect rowhammer attacks by monitoring for locality of DRAM memory accesses. To determine the locality of DRAM memory accesses, we use address sampling features provided by modern hardware performance counters.

In summary, the dissertation makes the following contributions:

Using pointer authentication for efficient temporal safety: In Chapter 2, we make the following contributions towards providing efficient temporal safety:

- We propose using pointer authentication for temporal safety. The proposed technique we call PETS adapts the lock-and-key mechanism, but instead of keeping per-pointer key metadata, it uses pointer authentication codes (PACs) embedded in the unused bits of pointers. We show that this allows using direct-mapped shadow to store metadata which improves performance compared to other lock-and-key techniques.
- We show how we can further improve the performance of PETS by using a low-fat memory allocator. In this scheme, the lock pointer is encoded in the pointer value itself, avoiding the need to store lock pointer metadata altogether.
- We evaluate PETS using SPECCPU2006 benchmarks and show that it has low runtime overhead compared to previous work (average overhead of 57% for low-fat PETS). In terms of memory overhead, we show that PETS has the least memory overhead

compared to other temporal safety techniques.

Using memory protection unit for efficient and strong sandboxing: In Chapter 3, we make the following contributions towards providing efficient sandboxing for embedded systems:

- We present uSFI, a code and data protection mechanism for low-resource devices. uSFI uses software analysis and widely available hardware support in embedded processors (memory protection units) to provide low-cost code and data isolation as well as I/O access control. Through the use of a specialized runtime and verifier, uSFI maintains these protections without instrumenting memory access instructions or indirect jumps, thus providing low-cost software module isolation even in the presence of buggy or malicious privileged code (e.g., a compromised kernel).
- We implement uSFI for the widely used ARMv7-M architecture. Using the MiBench embedded benchmarks suite and other real-world applications, we show that uSFI has low code size and performance overheads. Moreover, we show that the fraction of code that must be trusted in the system (i.e., the uSFI runtime) is a trivially small fraction of the overall system code size. At 150 lines of code, the attack surface of our trust management system can be easily analyzed and inspected to gain trust.

Using hardware performance counters to detect rowhammer attacks: In the last part of this thesis, we show that current rowhammer mitigation techniques for existing systems (i.e., disallowing cache flush instructions and doubling refresh rates) do *not* work. To that end, we proposed a software-based rowhammer detector that uses existing performance counter features. Specifically, we make the following contributions:

- We demonstrate the first *CLFLUSH-free* rowhammer attack, thereby thwarting efforts to deter rowhammering by restricting access to the *CLFLUSH* instruction.
- We present ANVIL, a software-based rowhammer detector which protects existing and future commodity DRAMs. We implement ANVIL using existing hardware performance monitoring infrastructure. ANVIL works by monitoring the locality of DRAM row accesses out of the last-level cache.

- We implement ANVIL as a Linux kernel module that utilizes Intel architecture performance monitoring capabilities to detect memory access locality. The detector uses a multi-staged approach to reduce detector overheads, leading to an average slowdown (for non-malicious programs) of about 1%, and worst-case slowdown of 3.2% for SPEC2006 integer benchmarks. The detector is accurate, with no false negatives and less than 1% false positives.

Dissertation Organization: The remainder of the dissertations is organized as follows. In Chapter 2, an efficient temporal safety technique is discussed. Chapter 3 proposes an efficient sandboxing mechanism for IoT-class devices. Chapter 4 details how hardware performance counters can be used to detect rowhammer attacks. Finally, we conclude in Chapter 5.

Chapter 2

Pointer Authentication for Efficient Temporal Memory Safety

2.1 Introduction

Temporal memory safety is one of the first line of defenses against memory corruption exploits. It is used to detect temporal errors such as use-after-free errors. Use-after-free errors occur when an object is accessed outside of the time during which it was allocated. The code example below shows a use-after-free vulnerability that can be used to mount a control-flow hijacking attack. The example is adapted from [10].

```
void(**ptr)() = malloc(sizeof(void*)); // Allocate space
*ptr = &func1;
...
void(**new_ptr)();
new_ptr = ptr;  // Copy pointer
...
free(ptr);     // Free space
user_input = malloc(...); // Reallocate space
*user_input = ... // Overwrite with input
(*new_ptr)();  // Use-after-free
```

Listing 2.1 Use-after-free Vulnerability Example

In the example *ptr* is a function pointer and points to the function *func1*. A copy of this pointer is made and is assigned to *new_ptr* on line 5. When the memory object is freed on

line 5, *new_ptr* becomes dangling. On line 8, the same memory object is reallocated and populated with a user-defined input value. The function call on line 10 creates a control-flow vulnerability because the address can be overwritten to an arbitrary value such as an ROP gadget.

Temporal memory safety techniques are used to detect such vulnerabilities. In this chapter, we show how modern processor features can be used to design an efficient temporal memory safety technique.

2.2 Background

In this section we introduce the concepts that are relevant throughout the chapter. We start by describing the *lock-and-key* technique - a popular temporal memory safety technique we build up on, and then we discuss two features that we use to improve the performance of the traditional lock-and-key technique.

2.2.1 The Lock-and-key Technique

A number of techniques known as temporal safety techniques have been proposed to protect against use-after-free vulnerabilities. The related work section discusses in detail the different kinds of temporal safety techniques. Here we will only describe one commonly used technique, the *lock-and-key* technique [9], which this work builds up on.Figure 2.1(a,b,c) demonstrates the lock-and-key technique. With this technique, each memory allocation (memory object) is assigned a lock, and each valid pointer to that allocation is assigned a matching key. On each access to the object, a check is made to make sure that the lock and key values match. When the memory is deallocated, the lock value is incremented by one so that checks will fail on subsequent references using dangling pointers.

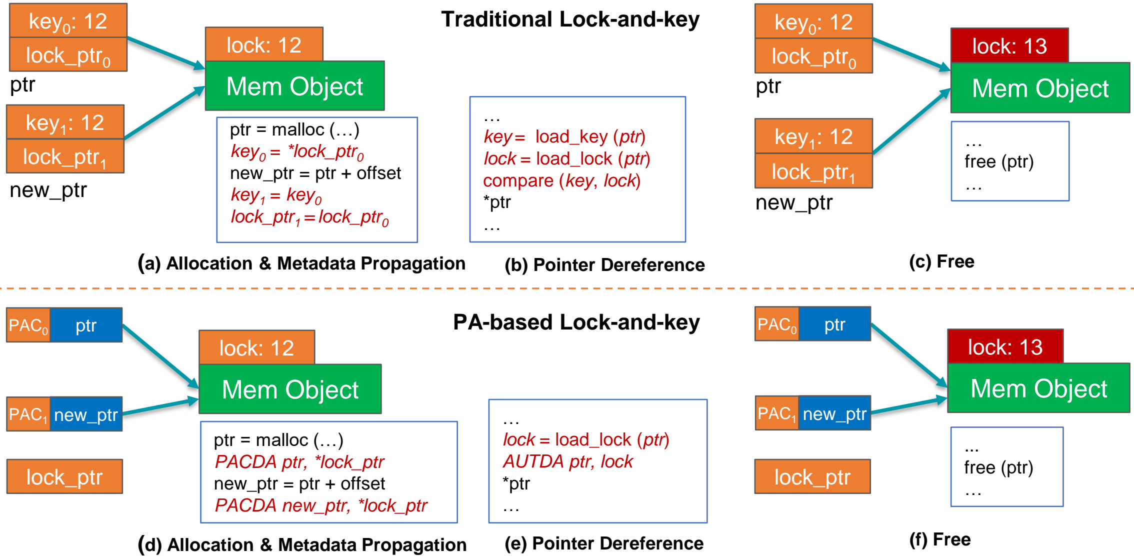


Figure 2.1 The Lock-and-Key Technique. The figure compares the *traditional* lock-and-key technique with our proposed pointer-authentication-based technique (PETS). When a memory object is allocated, it is assigned a lock. In the traditional technique, a matching key as well as a pointer to the lock is kept as metadata *for each* pointer. In our technique, pointer authentication code (PAC) values in the extra bits of pointers are used as a substitute to keys. In addition to this, only one lock address metadata is kept per the minimum allocatable object size. When a new pointer is derived from an old pointer, a PAC value is computed for the new pointer using the PACDA instruction. On each memory dereference, the PAC value of a pointer is verified using the AUTDA instruction. Finally, when the memory object is freed, the value of the lock is incremented by one.

2.2.2 ARM Pointer Authentication

In this work, we use pointer authentication to provide efficient temporal safety. Pointer authentication is a security primitive introduced in ARMv8.3-A and is used to verify the integrity of pointers [27]. It uses the unused bits in a pointer to store a cryptographic hash of the pointer value called *Pointer Authentication Code (PAC)*. The size of the PAC can vary from 11 to 31 bits depending on the system configuration. For example on AArch64, the Linux kernel uses a 39-bit virtual address space by default which allows for a 24-bit PAC [27].

The PAC is computed as a truncated output of the QARMA block cipher [47] or an implementation defined algorithm [48]. In this work we assume QARMA is used. As shown in Figure 2.2, three inputs are used to compute the PAC: the pointer value, a 128-bit secret

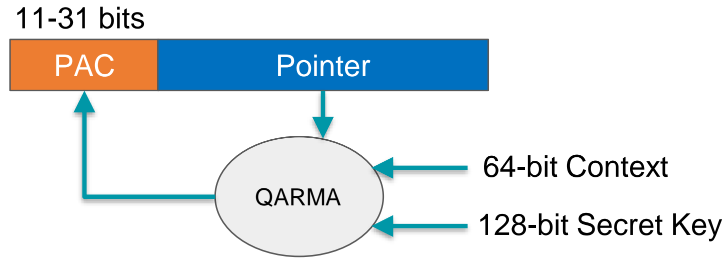


Figure 2.2 ARM Pointer Authentication – Computing the PAC. The PAC is computed as a truncated output of the QARMA block cipher.

---

key and a 64-bit *context* value. The key is stored in internal registers and is not accessible by user mode. The context value can be any user supplied value. In this work we use a memory object lock value as the context value. The Pointer Authentication specification defines five keys. Two keys (key A and key B) are used to compute PACs for data pointers and other two key are used for instruction pointers. A fifth key is provided for a generic authentication instruction.

The pointer authentication feature also provides instructions to compute and verify PAC values. For our work, we use two pointer authentication instructions: *PACDA* and *AUTDA*. The PACDA instruction computes and inserts a PAC for a data pointer using a context value and data pointer key A, while the AUTDA instruction authenticates a data pointer using a context value and data pointer key A [48]. If the authentication passes, the upper bits of the pointer are restored to enable subsequent use of the pointer. If the authentication fails, however, the upper bits are corrupted and any subsequent use of the pointer results in a translation fault.

2.2.3 The Low-fat Memory Layout

In Section 2.3 we show how metadata can be kept to the minimum by using a *low-fat memory layout*. The low-fat memory layout [49] subdivides a program's virtual address space into several regions. Each region is responsible for allocation of objects of a given fixed size range. Figure 2.3 shows an example low-fat memory layout. On the figure *region*

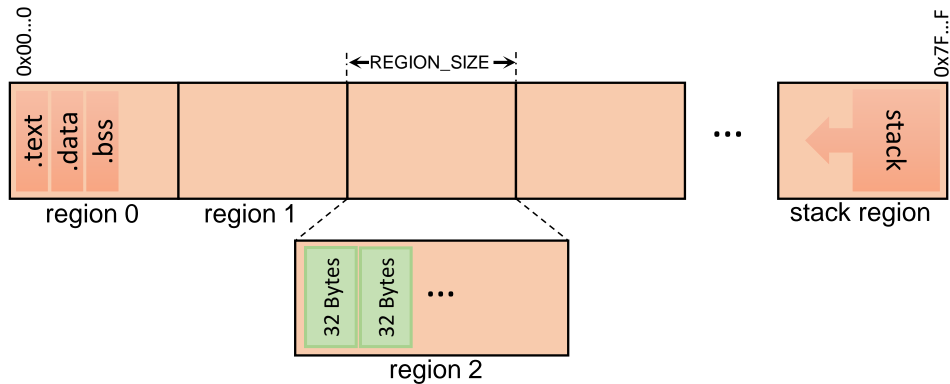


Figure 2.3 Low-fat Memory Layout. The low-fat memory layout subdivides a program's virtual address space into several regions. *Region 0* is used for text and data segments, and a *stack region* is used as program stack. Objects of a given size are allocated from a specific region. For example, objects of size 32 bytes are allocated from *region 2*.

---

0 is reserved for program *text* and *data* segments, and a special region (*stack region*) is assigned for the program stack. The rest of the regions contain sub-heaps for the *low-fat memory allocator*. The low-fat memory allocator allocates objects of a given size from a specific region. For example, suppose object sizes are restricted to be powers-of-two sizes, and the minimum allocatable size is 16 bytes. In this case objects of size 1-16 bytes are allocated from *region 1*, objects of size 17-32 bytes are allocated from *region 2* and so on.

The low-fat memory layout allows for implicit encoding of *object size* information in pointer values. The upper bits of a pointer indicate the region number which directly corresponds to the size of the object the pointer references. In addition, the base pointer of an object can be directly computed from any pointer to that object. This characteristics can be used for efficient bounds checking [49].

In the next section we will show how a low-fat memory allocator can be used to further reduce the overhead of our temporal safety technique.

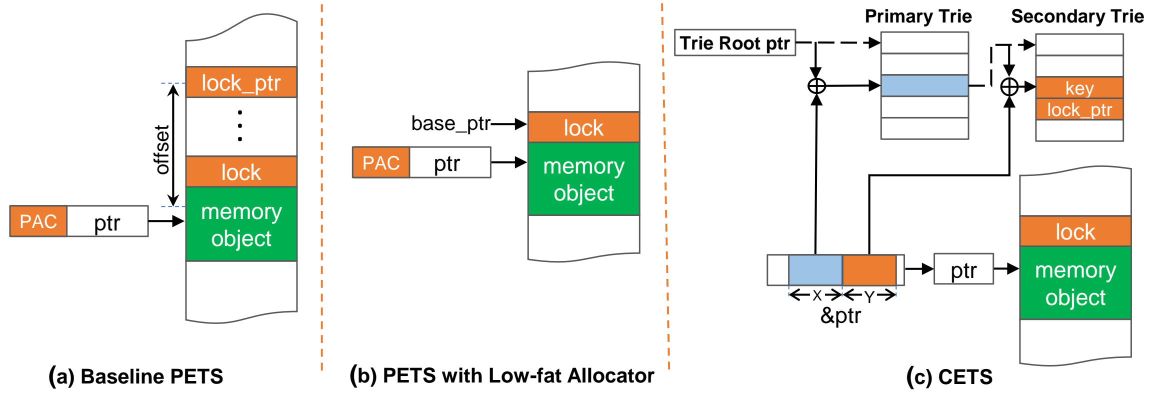


Figure 2.4 Metadata Storage and Access. The figure shows metadata storage for the two variants of PETS and CETS. The first technique is for baseline PETS where *one-level mapping* is used to map a memory object pointer to a lock pointer location. With this approach, two load operations are required to get the lock value. The second technique is for PETS with low-fat allocator. This technique doesn't require storing a lock pointer, as the lock pointer is the same as the base pointer, which is directly calculated from the pointer value. This approach requires only one load operation to read the lock value. The last technique is used by CETS [9], which we catagorize as traditional lock-and-key technique. CETS uses two-level lookup trie data structure to store metadata. With this approach each metadata access requires four loads.

2.3 Pointer Authentication for Temporal Safety

Though effective, the traditional lock-and-key technique described in Section 2.2 incurs large performance and memory overheads. In this work, we propose using pointer authentication to reduce the performance and memory overhead of the traditional lock-and-key technique. The proposed technique, which we call PETS, adapts the lock-and-key mechanism for temporal safety, but instead of using explicit storage for keys, we propose using PACs (Pointer Authentication Codes).

Allocation and Metadata Propagation: Figure 2.1 compares the traditional lock-and-key technique with PETS. Similar to the traditional lock-and-key technique, each memory allocation (memory object) is assigned a lock. When a pointer to that allocation is created (using a memory allocation function such as *malloc*) a PAC value is computed and stored in the unused bits of the pointer using the *PACDA* instruction (Figure 2.1(d)). This instruction takes the pointer value and a 64-bit context value as inputs. We use the lock value as a

context. When a new pointer is derived from an old pointer, for example through pointer arithmetic, a PAC value is computed for the new pointer.

Pointer Dereference: On each pointer dereference, the PAC value of a pointer is verified using the *AUTDA* instruction (Figure 2.1(e)). The AUTDA instruction computes a PAC using the pointer value and the lock value and compares it with the PAC value stored in the unused bits of the pointer. If the two PAC values do not match, the pointer is stale and corrupted so that a translation fault is generated if the pointer is dereferenced

Deallocation: Finally, when the memory is deallocated, for example using *free()*, the lock value is incremented, similar to the traditional lock-and-key technique (Figure 2.1(f)). On subsequent accesses using a dangling pointer, PAC verification should fail since the lock value has changed.

We explored two variants of PETS: PETS with shadow storage (we will refer to this as *baseline PETS* for the rest of the paper) and low-fat PETS. Both variants use pointer authentication; the only difference is the memory allocators they use. We discuss the two variants in detail below.

2.3.1 PETS with Shadow Storage (Baseline PETS)

Baseline PETS uses the standard C library memory allocator (dlmalloc-based allocator). Each heap memory object, regardless of the size of the object, keeps a 64-bit lock value. In addition to this, a 64-bit pointer to the lock (lock pointer) is kept per the minimum allocatable heap object. In our implementation the minimum size of a heap object is 32 bytes. Figure 2.4(a) shows how metadata is stored and accessed in the baseline PETS implementation. We use *one-level mapping* to map a pointer value to its metadata (lock pointer) location. With this mapping, the lock address is located at a fixed offset from the pointer value. This way the lock pointer location can be calculated by a single shift and

addition operations. On memory allocation, the memory allocator allocates metadata space for the memory object if it is not already allocated and updates the value of the lock pointer that corresponds to the new pointer.

The PETS compiler adds code to propagate metadata (including computing PAC) whenever a new pointer is derived from an old pointer. The code below shows the operations performed on pointer propagation. The lock pointer is copied from the old pointer lock pointer location to the new pointer lock pointer location. Then a PAC is computed for the new pointer after loading the lock value.

```
// Copy lock pointer
lock_ptr_addr = compute_lock_ptr_address(old_ptr)
old_lock_ptr = *lock_ptr_addr
lock_ptr_addr = compute_lock_ptr_address(new_ptr)
*lock_ptr_addr = old_lock_ptr
// Compute PAC
lock = *old_lock_ptr
PACDA new_ptr, lock
```

Listing 2.2 Baseline PETS - Metadata Propagation Code

The PETS compiler also adds code to verify PAC values on every relevant load and store as shown on the code below. The verification code first loads the lock that corresponds to the pointer from the metadata location. If the lock matches the lock used during PAC computation, then the verification should pass. When the memory object is freed, the memory allocator increments the lock value.

```
// Load lock and verify PAC
lock_ptr_addr = compute_lock_ptr_address(ptr)
lock_ptr = *lock_ptr_addr
lock = *lock_ptr
AUTDA ptr, lock
```

Listing 2.3 Baseline PETS - PAC Verification Code

2.3.2 Low-fat PETS

The baseline technique can be further optimized if a low-fat memory allocator is used instead of a dlmalloc-based allocator. In this technique, similar to the baseline PETS, each heap memory object keeps a 64-bit lock value at the *base* of the memory object (*i.e.* at the base pointer). With low-fat memory layout the base pointer of a memory object can be calculated from a pointer value, therefore there is no need to keep a lock pointer metadata. Figure 2.4(b) shows how metadata is stored and accessed for the low-fat based PETS implementation. For our implementation we used powers-of-two sizes for the heap objects with a minimum heap object size of 16 bytes. With this implementation, the base pointer can be calculated by fast logical shift and AND operations. The code below shows the operations for metadata propagation. Unlike baseline PETS, there is no need to copy metadata during propagation: only PAC computation for the new pointer is performed.

```
base_ptr = compute_base_pointer(new_ptr)
lock = *base_ptr
PACDA new_ptr, lock
```

Listing 2.4 Low-fat PETS - Metadata Propagation Code

In addition to this, on pointer dereference, the lock value is directly loaded from the base pointer, requiring only one load operation to read the lock value.

```
base_ptr = compute_base_pointer(ptr)
lock = *base_ptr
AUTDA ptr, lock
```

Listing 2.5 Low-fat PETS - PAC Verification Code

As a comparison with the closest related work, Figure 2.4(c) shows the metadata storage technique used by CETS [9], which we catagorize as traditional lock-and-key technique. CETS uses two-level lookup trie data structure to store metadata. It stores key and lock pointer metadata *for each* pointer. The upper x bits of a pointer address are used to index

the primary trie structure and the lower y bits are used to index the secondary trie structure. With this approach each metadata access requires four loads as opposed to only one load operation for low-fat PETS.

As we will show in Section 2.5, the metadata storage and access technique significantly affects the performance of the lock-and-key technique.

2.4 PETS Implementation

In this section, we discuss the implementation details of PETS. We describe the two components of PETS: the PETS compiler instrumentation and the memory allocators.

2.4.1 PETS Compiler Instrumentation

PETS instruments code as described in Section 2.3. We implemented the compiler instrumentation with an LLVM [50] pass we call *UAF detector pass*. Algorithm 2.1 shows the algorithm for UAF detector pass. The pass checks each instruction to determine whether it is a pointer arithmetic instruction or a memory access instruction. For a pointer arithmetic instruction, it inserts the code on Listing 2.2 or Listing 2.4 *after* the instruction, depending on the variant of PETS used. Similarly, for a memory access instruction the code on Listing 2.3 or Listing 2.5 is inserted *before* the instruction.

Like mentioned earlier, stack use-after-free errors are rare, therefore PETS targets heap use-after-free errors. For this reason, no instrumentation code is inserted if a pointer is statically (at compile time) determined to point to the stack or global data. However, it is not always possible to determine where a pointer points to at compile time, therefore code that dynamically checks whether the pointer is within the bounds of the heap is inserted at the beginning of the instrumentation code. In a typical address space layout, the heap is located above the *.bss* section and below the stack, therefore the rest of the instrumentation code is bypassed if the pointer is greater than or equal to the stack pointer, or it is less than the end

Algorithm 2.1 Algorithm for UAF Detector Pass

```
 1: procedure INSTRUMENTFUNCTION(Function F)
 2:     for Instruction I in F do
 3:         if pointerArithmeticInst (I) then
 4:             ptr ← getSrcPtr(I)
 5:             if isGlobalVar (ptr) —— isStackVar (ptr) then
 6:                 continue
 7:             instrumentPtrArithmetic (I)            ▷ Listing 1 or 3
 8:         else if memoryAccessInst (I) then
 9:             ptr ← getAddress(I)
10:             if isGlobalVar (ptr) —— isStackVar (ptr) then
11:                 continue
12:             instrumentMemoryAccess (I)             ▷ Listing 2 or 4
```

of the *.bss* section (bss_end) as shown below. PETS doesn't need to know where the start or the end of the heap is, therefore it is compatible with ASLR.

```
// Bypass check if pointer points to data
// in the stack or data section
if (ptr >= stack_ptr)
  goto end
if (ptr < bss_end)
  goto end
// Rest of instrumentation code
end:
```

Compiler Optimizations

The PETS compiler eliminates redundant pointer dereference checks similar to [9]. Given a memory access instruction I that uses a pointer, if the instruction is dominated by an earlier check of the pointer and there is no call to a *free()* function between I and the check, then no check is inserted before I.

The PETS compiler doesn't perform inter-procedural analysis to determine calls to *free*, instead we conservatively assume any function call would potentially free any memory object. With this approach we were able to remove a significant number of redundant check

on some benchmarks as shown on Figure 2.5.

2.4.2 Memory Allocators

As detailed in Section 2.3, we evaluated two variants of PETS. The two variants use different memory allocators.

Baseline PETS: The baseline PETS uses a dlmalloc-based memory allocator. For our implementation, we modified the musl C library memory allocator. Specifically, we made the following modifications:

- The *chunk* struct which maintains metadata for memory chunks (memory objects) was modified to include a *lock* element.
- Memory allocation functions (*malloc, calloc, realloc*) were modified to include a routine that allocates metadata space and initializes the *lock pointer* value in the metadata space.
- On *free*, the *lock* value for the corresponding chunk is incremented by 1.

During the memory allocation or freeing process, memory objects can be split or merged together. The value of the lock in the memory object needs to account for these operations. When a memory object is split into two, the new memory object inherits the lock value of the parent memory object. When two memory objects are merged together, if the two objects have different lock values, the *larger* lock value is taken as the lock value of the new object. This insures that a lock value is never reset back to an older value.

Low-fat PETS: The low-fat PETS implementation uses a low-fat memory allocator. We used a modified version of a low-fat allocator implementation provided at [51]. We configured the allocator to use powers-of-two sizes. There are 30 possible heap object sizes with a minimum allocation size of 16B and a maximum size of 8GB. There are 32 regions in total including the *stack region* and *region 0* which is used for code and data sections as shown in Figure2.3. In our implementations, we assume a 39-bit virtual address space which is

the default for the Linux kernel on AArch64 [27]. The most significant 5 bits are used to indicate the memory object size. This leaves 34 bits to index a region.

With this setting the *compute_base_pointer* function on Listings 2.4 and 2.5 is implemented as:

```
compute_base_pointer (ptr) {
size = ptr >> 34
mask = 0x7FFFFFFF8 << size
base_ptr = ptr & mask
return base_ptr
}
```

We made the following modifications to the memory allocator to enable use-after-free protection:

- A *lock* value is added at the base address of each allocation.
- On *free*, the *lock* value is incremented by 1.

2.4.3 Support for Multithreaded Applications

Multithreaded applications present a challenge for memory safety techniques. One issue is a potential data race when accessing shared metadata structures. For example, CETS uses a shared *next_key* counter that is incremented whenever a memory object is allocated [9]. To avoid races, either accesses to the counter need to be synchronized or each thread need to keep its own counter. PETS doesn't have such issues because metadata is kept for each memory object.

However there are situations where temporal safety might be violated if the multithreaded application is not properly synchronized. For example, between an address check and a memory object access, another thread might free the memory object, resulting in missed violation. PETS supports multithreaded programs without any change as long as the programs are properly synchronized (*i.e.* they are data-race free).

2.5 Experimental Evaluation

In this section we discuss the evaluation of PETS. We evaluated the runtime and memory overheads of the two variants of PETS and compared them with other temporal safety techniques. We start by detailing our setup for the evaluations.

2.5.1 Experimental Setup

Pointer Authentication Implementation

PETS makes use of the ARM pointer authentication extension. ARM pointer authentication extension is a feature of the new ARMv8.3-A ISA specification. At the time of the writing of this paper there were no hardware implementations that support ARMv8.3-A, therefore we had to model the pointer authentication feature. Specifically we modeled the two instructions used by our technique (*i.e.* PACDA and AUTDA instructions).

Correctness Evaluation: To evaluate the correctness of our technique (*i.e.* to verify that programs behave correctly with PETS instrumentation), we modeled the two instructions in the QEMU emulator [52]. We ran SPECCPU2006 benchmarks and verified the correctness of the outputs.

Runtime Overhead Evaluation: To evaluate runtime overhead, we modeled PACDA and AUTDA instructions in the gem5 cycle-accurate simulator [53]. In our gem5 model we assumed there is a single separate functional unit that performs pointer authentication related operations such as computing a PAC value as shown in Figure 2.2 and comparing the result of a PAC computation with a PAC value stored in a pointer. We modeled the PAC computation to take 8 cycles at a CPU frequency of 1.5 GHz based on the results of QARMA encryption latency given at [47].

Our tests using the model in gem5 revealed that the contribution in runtime overhead of

PAC computation and verification is negligible compared to the rest of PETS instrumentation code (For example, on baseline PETS, there are two loads per memory access check). For this reason and because running large benchmarks (such as SPECCPU2006 benchmarks) to completion in a cycle-accurate simulator takes a very long time, runtime evaluations are done on a real hardware by replacing PACDA and AUTDA instructions with NOPs.

PAC Size: In our implementations, we assume a 39-bit address space. That means there are 24 bits for PACs. Each memory object has a separate 64-bit lock value which is incremented whenever the memory object is freed. However due to the limited number of PAC bits we essentially have a 24-bit entropy. It is possible to get collusions during the lifetime of a memory object. The PETS memory allocator stops reallocating a memory object when the lock value overflows.

Evaluation Platform

As mentioned earlier, our model of the pointer authentication instructions in the gem5 simulator showed that the effect of pointer authentication instructions on the performance of the overall system is negligible. Therefore, all the evaluations in this section are performed on a real hardware. Note that only runtime is affected by pointer authentication instructions - memory overhead is not affected by the instructions.

Specifically we used the Xilinx ZCU102 evaluation board [54] for our evaluations. The board includes four ARM Cortex-A53 cores with 32kB L1 instruction and data caches and a 1MB shared L2 cache. The board also includes a 4GB DDR4 DRAM.

Benchmarks: We used the C benchmarks from SPECCPU2006 for our evaluations. We used the default *-O2* optimization when compiling the benchmarks.

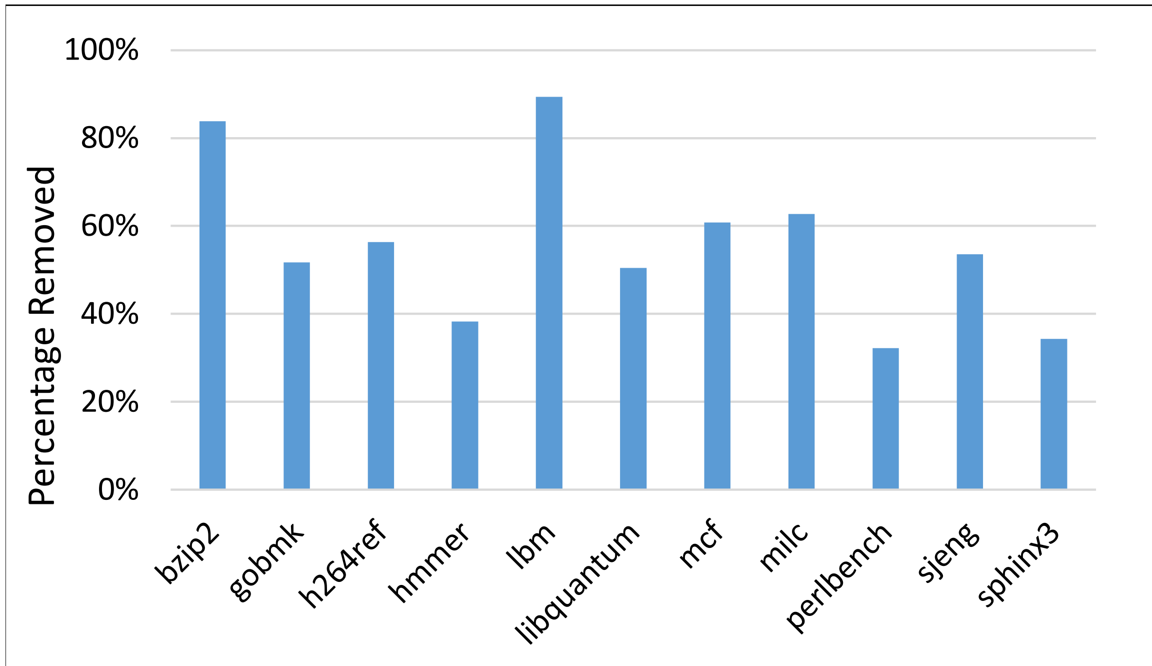


Figure 2.5 Percentage of Redundant Memory Access Checks Removed by PETS Compiler Optimization.

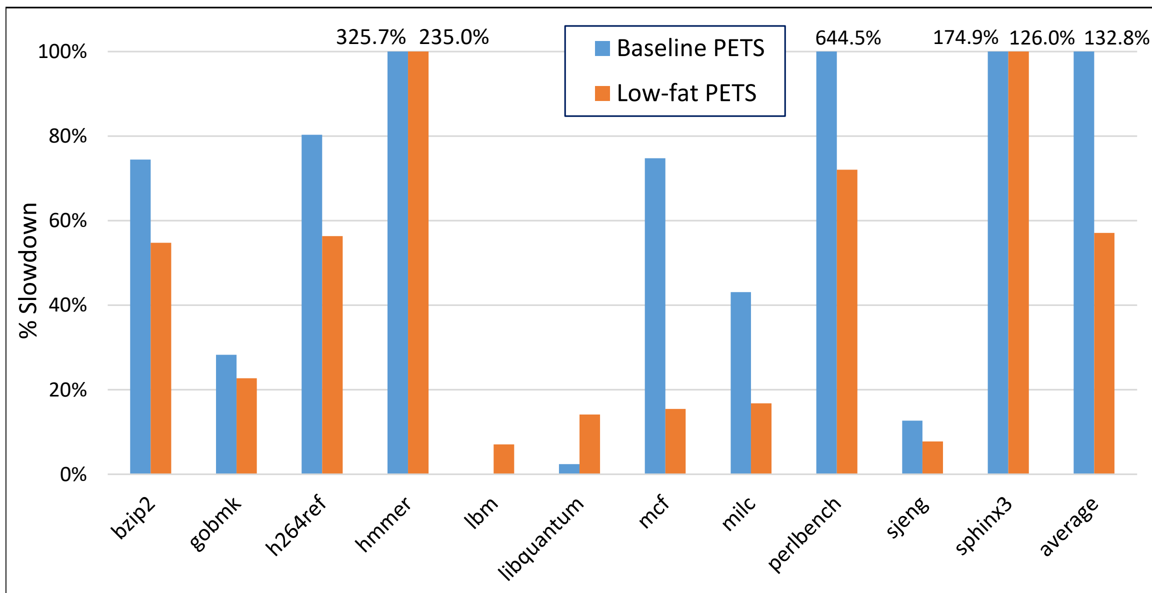


Figure 2.6 PETS Runtime Overhead. The graph shows the runtime overhead of the two variants of PETS for SPEC2006 benchmarks. In almost all benchmarks low-fat PETS performs better than baseline PETS. Most of the overhead for PETS comes from memory access checks. We can see a correlation between the overhead and the percentage of redundant checks removed shown on Figure 2.5. The benchmarks with the highest overhead are the ones with the least optimization.

2.5.2 Runtime Overhead:

We measured the runtime overhead of the two variants of PETS. For the evaluation, the benchmarks were compiled with the redundant check optimization discussed in Section 2.4. Figure 2.5 shows the percentage of redundant memory access checks removed by the optimization. As shown in the figure, the optimization removed a significant number of redundant checks.

Figure 2.6 shows the runtime overheads of the two variants of PETS as compared to a baseline system without PETS instrumentation. As shown in the figure, lowfat-based PETS performs better on all the benchmarks except *libquantum* and *lbm*. This is attributed to the smaller number of loads per memory access checks for lowfat-based PETS. The two exceptions are due to the overhead of the low-fat memory allocator outweighing the overhead of the instrumentation. We can see a correlation between the percentage of redundant checks removed and the runtime overhead. Benchmarks that have the highest overheads (*perlbench*, *hmmer* and *sphinx3*) are also the ones with the least optimization. In summary, low-fat PETS has an average overhead of 57.12% while baseline PETS has an average overhead of 132.86%.

2.5.3 Memory Overhead

We also evaluated the physical memory overhead of PETs. Figure 2.7 shows the memory overhead of the two variants of PETS. The memory overhead was calculated using readings from the */proc/pid/status* file in Linux by adding the peak resident set size (vmHWM), page table entries size (VmPTE) and size of second-level page tables (VmPDE).

As shown on the figure, low-fat based PETS has a very low memory overhead averaging at 1.9%. Most of the overhead for low-fat PETS comes from the low-fat allocator itself. Baseline PETS has an average overhead of 28.4%. This is expected as 8-byte lock pointer metadata is kept for every 32-byte memory object. Ideally, this would result in a memory

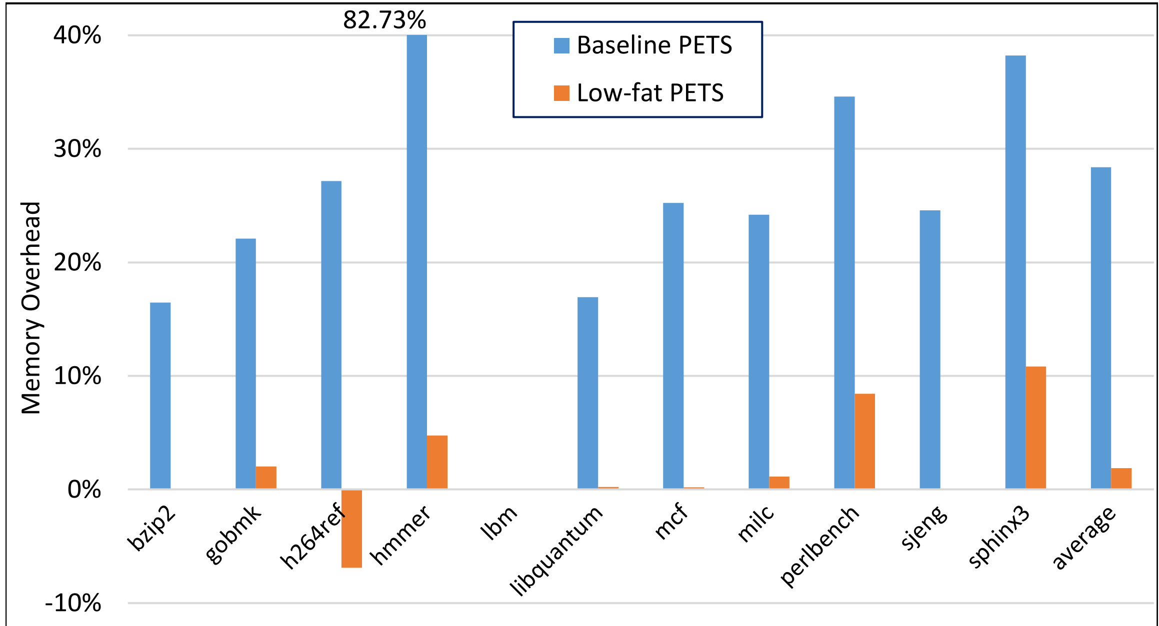


Figure 2.7 PETS Memory Overhead. The graph shows the physical memory overhead for the two variants of PETS. From the graph we can see that low-fat PETS has a very low memory overhead as it almost entirely avoids using metadata. Note that h264ref has negative overhead for low-fat PETS. This is attributed to the low-fat memory allocator being more efficient than the baseline dlmalloc-based allocator.

---

overhead of 25%. The extra overhead comes from addition of a lock metadata per allocated memory object and the associated object alignment requirement.

2.5.4 Comparison with other Temporal Safety Techniques

The related work section discusses in detail the pros and cons of related temporal safety techniques, here we compare performance of PETS with some of these techniques. All the other techniques were implemented and evaluated on an X86 system and on different microarchitecture than ours, therefore we don't do a direct comparison of runtime overhead. However, we do a direct comparison of memory overhead as it is independent of architecture.

Comparison with CETS: CETS [9] is a lock-and-key technique that uses a disjoint metadata per pointer as shown on Figure 2.4. It is the closest work to PETS. CETS needs four load operations per metadata access, while lowfat-PETS needs only one, therefore CETS is

expected to have a higher runtime overhead as compared to PETS.

Table 2.1 compares memory overhead of low-fat-based PETS with CETS. The original CETS paper doesn't provide memory overhead results. We estimated the overhead of CETS from the results given at the follow-up work SoftBoundCETS [17]. SoftBoundCETS provides both temporal and spatial memory safety. It keeps four 64-bit values (lock, key, base and bounds) as metadata per pointer while CETS keeps only two of the values (lock and key). Therefore we estimated that temporal protection alone (CETS) contributes to at least half of the memory overhead of SoftBoundCETS. The numbers for CETS in Table 2.1 are half of that is given for SoftBoundCETS at [17]. The numbers for some of the benchmarks are left blank since data is not available. From the table, we can see that low-fat PETS has far lower memory overhead than CETS on almost all the benchmarks. The average overhead for CETS for the benchmarks where data is available (92%) is much higher than even the baseline PETS (28.4%).

Comparison with DangSan and DangNull: Another temporal safety technique keeps track of all pointers to a memory object. When the memory object is deallocated, the values of all pointers to that object are corrupted by for example setting the values to NULL. DangSan [11] and DangNull [12] are such two techniques.

These techniques do not need to instrument pointer dereferences as the memory managment unit (MMU) performs null checks in hardware. The overhead for these techniques comes from traversing pointer tracking data structures during allocation and deallocation of memory objects. This means these techniques have better runtime performance than PETS on applications that do not allocate/deallocate memory frequently. However for benchmarks that create 100s of millions of objects, such as *perlbench*, we expect PETS to perform better.

In terms of memory overhead, as shown on Table 2.1, DangNull's and DangSan's overhead vary greatly depending on the benchmark, but overall lowfat-based PETS performs much better (1.9% average overhead for low-fat PETS vs. 212% average for DangNull and

Benchmark	Memory Overhead (%)				
	Low-fat PETS	DangNull	DangSan	Oscar	CETS
bzip2	0.07	0	0	0	-
gobmk	2.04	21	120	0	250
h264ref	-6.90	373	16	10	190
hmmer	4.75	1700	30	33	-
lbm	0	0	1	0	40
libquant	0.23	0	2	5	0
mcf	0.20	0	62	0	-
milc	1.14	0	25	16	55
perlbench	8.44	-	380	225	-
sjeng	-0.02	0	2	0	0
sphinx3	10.82	34.7	180	400	110

Table 2.1 Memory Overhead Comparison. The table compares the physical memory overhead of low-fat PETS with other temporal safety techniques. From the table we can see that low-fat PETS has by far the least overhead.

74% average for DangSan).

Comparison with Oscar: Oscar [10] is a recently proposed page-permission-based temporal safety technique. Oscar doesn't instrument code - its overhead originates from creating shadow pages during allocations. For this reason, Oscar has low runtime overhead for applications that do not allocate memory frequently. However, PETS has much lower memory overhead than Oscar (1.9% average for lowfat-PETS vs. 62% average for Oscar for the benchmarks on Table 2.1).

In summary, low-fat PETS by far has the least memory overhead compared to the other temporal safety techniques on Table 2.1. The runtime overhead of PETS is expected to be lower than other lock-and-key techniques such as CETS.

2.6 Related Work

In this section we review previous work, focusing on temporal safety techniques.

Technique	UAF Detection Mechanism				Metadata Managment					
	Lock-and-key	Dangling Pointer Tagging	Page-permission-based	Probablistic/ Memory-reuse Delay	Direct-mapped Shadow	Multi-level Shadow	Custom Data Structure	Fat Pointers	Tagged Pointers	Disjoint Per-pointer Metadata
CETS [9], Watchdog [18]	✓									✓
Safe-C [14]	✓							✓		✓
W. Xu, et.al [25]	✓									✓
Undangle [55]		✓								✓
FreeSentry [13], DangSan [11]		✓				✓	✓			
DangNull [12]		✓					✓			
Intel Pointer Checker [56]		✓								✓
Electric Fence [57], PageHeap [58]			✓							
D. Dhurjati et al [59], Oscar [10]			✓							
Purify [60]				✓	✓					
Dr. Memory [61]				✓		✓				
Memcheck [62]				✓		✓				
DieHard [63], DieHarder [64]				✓						
Baseline PETS	✓				✓				✓	
Low-fat PETS	✓								✓	

Table 2.2 Use-after-detection Mechanism and Metadata Management for Temporal Safety Techniques.

2.6.1 Temporal Safety Techniques

Table 3.6 categorizes the different temporal safety techniques based on the technique used to provide protection against use-after-free errors and the type of metadata management used. The categories are based on [26]. Below we discuss about the pros and cons of each temporal safety and metadata management technique.

Lock-and-key: PETS is based on the lock-and-key temporal safety technique. The lock-and-key technique can detect all use-after-free errors. The performance of a lock-and-key techniques depends on the type of metadata management used. Safe-C [14] and W. Xu et al [25] use fat pointers to store unique capabilities (similar to keys). CETS [9] and Watch-Dog [18] use disjoint metadata space to store per-pointer lock and key. Both techniques require a change in the calling convention if a pointer is used as an argument to a function, as metadata need to travel with the pointer. With PETS on the other hand, PACs are embedded in pointers, therefore there is no need to transform function declarations. In addition to this, the use of per-memory-object metadata as opposed to per-pointer metadata makes PETS more efficient than other lock-and-key techniques.

Dangling Pointer Tagging: One way of detecting dangling pointer dereferences is to keep track of all pointers to a memory object and to corrupt the pointers, for example by nullifying them, when the object is freed. Then subsequent dangling pointer dereferences would result in address translation fault. Undangle [55], FreeSentry [13], DangNull [12] and DangSan [11] use this technique. These techniques add instrumentation in code that maps a newly created pointer to a memory object. FreeSentry and Dangsan use a two-level lookup table to map the newly created pointer to a memory object. Then they record the address of the pointer in a linked list associated with the memory object. When an object is freed, the linked list associated with object is traversed to invalidate the pointers to that object. DangNull is a similar techniques, but instead of multi-level shadow, it uses red-black tree

to track point-to relations. These techniques do not need to instrument pointer deferences, therefore have lower runtime overhead compared to lock-and-key techniques. However, on applications that perform many allocations, their overhead can be high. Furthermore, as shown in Table 2.1, the techniques are very memory intensive, especially on applications that perform many allocations.

Page-permissions-based: Another temporal safety technique involves assigning new virtual and/or physical pages for each allocation. On deallocation, the permissions on the virtual pages are changed so that dangling pointer dereferences are detected through page faults. The naive approach of assigning new physical pages for each allocation used by Electric Fence [57] and PageHeap [58] has impracticality high physical memory overhead. To get around this problem, D. Dhurjati et al [59] proposed using aliased virtual pages. In this scheme, a new virtual page for each allocation is used, but instead of using a new physical page, same physical page as the original program is used. On deallocation, individual virtual pages corresponding to the freed object are disabled and are never remapped. One drawback of this scheme is that applications with many allocations can exhaust physical memory due to accumulation of data structures that the operating system maintains for disabled virtual pages. Oscar [10] proposed using ”high water mark” when creating virtual shadows and unmapping old shadows when a memory object is freed. This reduces the amount of accumulated data structures.

Page-permission-based schemes in general do not require code instrumentation and do not keep explicit metadata, therefore have better compatibility and low runtime overheads. However, similar to dangling pointer tagging, allocation intensive applications can have high runtime and memory overheads.

Probabilistic/ Memory-reuse Delay: Randomized memory allocators [63, 64] places memory objects randomly across a heap such that the probability of a newly-free object being reallocated is low. Probabilistic techniques can be attacked by deliberately making

large allocations, therefore forcing reuse of freed objects.

Some memory error detection tools [60, 62, 65, 61] detect use-after-free errors by delaying (putting the object in quarantine) reallocation of memory objects after they are freed. They detect accesses to *quarantined* objects by disabling access to the objects or filling the objects with patterns and looking for those patterns when memory is accessed. Detection accuracy of delay-based techniques depends on how long a freed object stays in quarantine, and do not guarantee detection of all use-after-free errors.

2.7 Chapter Summary

In this chapter, we presented PETS, a temporal safety technique that utilizes authenticated pointers to reduce the space used for metadata storage. PETS adopts the lock-and-key temporal safety technique, but instead of storing key metadata for a pointer in memory, PETS stores it in the unused bits of the pointer itself. This reduces the total metadata stored in memory allowing for fast metadata access. PETS has better compatibility and lower memory and runtime overheads compared to previous lock-and-key techniques, and has the least memory overhead compared to other temporal safety techniques. PETS thereby brings the lock-and-key technique a step closer to being used as a practical runtime use-after-free detection mechanism.

Chapter 3

Efficient Software Fault Isolation for IoT-Class Devices

3.1 Introduction

Embedded devices are everywhere, with low-end embedded devices being used in many critical systems such as medical, industrial, and automotive applications. With the Internet-of-Things (IoT) promising as much as 30 billion connected devices by 2020, the security of low-end embedded devices has become a major concern. The rising number of devices along with increased connectivity has immensely exacerbated the attack surface of this class of devices, making them a genuine target of interest for saavy attackers. Two recent examples of this trend include the Jeep remote kill attack [66] and the Mirai IoT-based botnet [67].

While the exposure to attacks is increasing, the security mechanisms in these systems remain mediocre due to the tight resource (*e.g.*, computing power and memory size) and price constraints. Recent works have shown that embedded devices suffer from the same memory corruption vulnerabilities that have plagued traditional computing systems [28, 30, 29]. In [30] a stack overflow vulnerability inside an ARM Cortex-R4 processor embedded in a Wi-Fi chip was used to execute arbitrary code and ultimately take over a mobile device by Wi-Fi proximity alone.

Low-end embedded systems typically operate in a single address space with tightly-coupled software components (*e.g.*, RTOS kernels, peripheral drivers, libraries, *etc.*) without any form of isolation between software components. These software components often

come from different chip, sensor and I/O device vendors, which ultimately raises serious questions as to whether or not these devices can be trusted. Software modules might contain bugs or even malicious code that could be used to compromise the security of the whole system. Consequently, there is a great need to provide IoT-class devices with low-cost, reliable protections against widely employed attacks, such as memory corruption and code injection.

In this chapter, we leverage the rudimentary memory protection support found in modern IoT-class microcontrollers to build a low-profile, low-overhead, flexible sandboxing mechanism that can provide isolation between tightly-coupled software modules. With our approach, named uSFI, only the trust management code need to be trusted. Through the use of a static verifier and monitored inter-module transitions, module code at all privilege levels (including the kernel) is able to run uninstrumented and untrusted code. We implemented uSFI on an ARMv7-M based processor, both bare metal and running the freeRTOS kernel, and analyzed the performance using the MiBench embedded benchmark suite and two additional highly detailed applications. We found that performance overheads were minimal, with at most 1.1% slowdown, and code size overheads were also low, at a maximum of 10%. In addition, our trusted code base was trivially small at only 150 lines of code.

The remainder of this chapter is organized as follows. In Section 3.2, we outline the threat model for this work. Section 3.3 discusses the details of uSFI system architecture. Section 3.4 details our implementation of uSFI for the ARMv7-M architecture. In Section 3.5, we evaluate uSFI using representative embedded benchmarks and other real-world applications. In Section 3.6, we discuss related work in the area of software-based fault isolation and embedded device security, and finally we draw conclusions in Section 3.7.

3.2 Threat Model

The goal of uSFI is to protect the code and data of software modules from being leaked or corrupted by other compromised or malicious modules. In a uSFI-enabled system, software modules (library code, drivers, etc.) are untrusted (*i.e.*, they might contain software bugs or malicious code that lead to compromised execution). Modules can execute any code within their sandbox, including attacker selected code gadgets [6, 68].

A compromised module will try to execute attacker code by overwriting code memory, executing data, or forming gadgets out of existing code. Furthermore, a corrupted module will try to corrupt the data memory or read sensitive data such as encryption keys from the memory of other modules. Finally, a compromised module will try to gain elevated access to I/O to gain access to a remote controller or leak sensitive information.

Unlike other low-end fault isolation systems, we assume the kernel is not trusted. Similarly, the system compiler is not trusted, instead a trusted verifier is used to verify module code generated by the compiler. The uSFI verifier and the uSFI runtime are the only trusted software components. We assume the underlying hardware is trusted. Finally, we also assume there is a trusted bootloader that verifies and loads binaries at system startup.

3.3 uSFI System Architecture

The goal of uSFI is to protect code and data from untrusted software modules. Untrusted software modules include core modules, third party libraries, drivers, and operating system kernels. This is achieved by sandboxing modules in their own security domain and using a well-defined interface for cross-domain procedure calls. In this section, we provide details of the uSFI architecture.

A uSFI-enabled system has two components: a trusted runtime and untrusted modules. The uSFI runtime is the only trusted component in the system, and it has access to the entire memory and sole access to the memory protection resources. Software modules, including

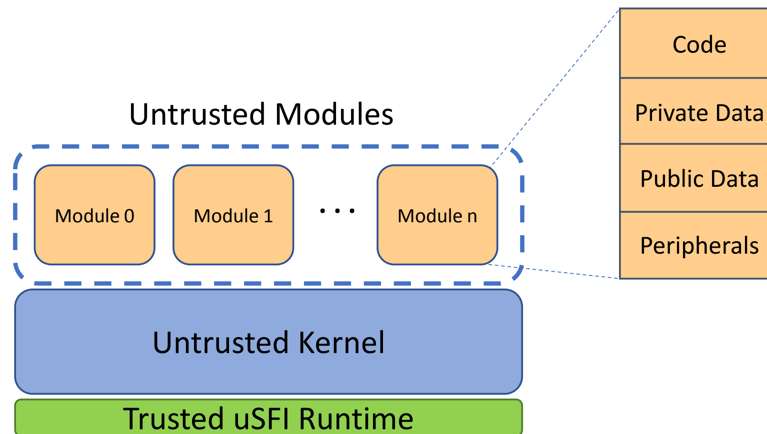


Figure 3.1 uSFI System Components. A uSFI-enabled system is composed of a trusted runtime and sandboxed untrusted modules. Modules might include drivers, libraries and user applications. In an embedded system that uses an RTOS kernel, the kernel is also considered untrusted (*i.e.*, it can not read/write system configuration and control resources, and modules' memory). Each module has its own code and data regions. A module can only access its own data or other modules' public data. In addition to this, a module can only access peripherals assigned to it. Sandboxing of modules is facilitated by the use of readily available Memory Protection Unit (MPU) hardware. An MPU allows partitioning memory into regions and assigning attributes and access permissions to each region.

the kernel and drivers, are all untrusted. A module represents a single security domain with its own code, data and peripheral memory regions. Figure 3.1 illustrates this isolation capability. A module can only execute code that resides in its code region, and it can only access data belonging to itself or public data in other modules. Furthermore, a module can only access peripherals assigned to it.

3.3.1 Module Privilege Levels

Typically code running at privileged level (*e.g.*, kernel code) has access to all system resources including system control and configuration resources. In a uSFI-enabled system, however, even a privileged code module is sandboxed such that it only has access to memory regions assigned to it. This is achieved by adding a third privilege level, *restricted-privileged level*, in addition to privileged and unprivileged levels. In the restricted-privileged level, a module has access to privileged instructions, but its memory access is still restricted such that it cannot compromise the uSFI runtime. As such, it has no access to the memory

management resource as well as other modules' memory regions.

There are currently multiple ways to achieve this restricted-privileged access level. The first method is to augment memory load and store instructions inside a privileged module with address check instructions. Another method is to force all untrusted software to operate at the unprivileged level and call trusted privileged code when privileged operations are needed. Unfortunately, both of these methods incur significant performance overheads. uSFI, on the other hand, uses a novel approach that takes advantage of *unprivileged memory access instructions*. With this approach, a privileged module is forced to use unprivileged memory access instructions that grant access to only the memory that the uSFI runtime permits it to access, instead of allowing carte blanche memory access with privileged loads and stores. These unprivileged instructions perform the same operations as privileged load and store instructions except that the MPU continues to enforce uSFI delineated memory access domains. This method incurs negligible performance overheads.

Through the use of a static binary verification mechanism, the restricted-privileged module (*e.g.*, an RTOS kernel) is not allowed access to memory configuration registers; however, it still can perform many privileged tasks without requiring the trusted uSFI runtime's help. For example, in the ARMv7-M architecture, a widely used architecture in embedded devices, a restricted-privileged module will have access to system instructions (MSR and MRS instructions). These instructions are used to read and write special purpose registers. For example, use of these instructions can enable, disable or change interrupt priorities through access to the BASEPRI register. There are, however, a few cases where a restricted-privileged module needs the help of the uSFI runtime. In particular when context switching between tasks. In such cases, the module issues a supervisor call to the runtime to request the required service.

To summarize, there are three privilege levels in a uSFI-enabled system: privileged, unprivileged and restricted-privileged. Only the uSFI runtime runs at the privileged level. Modules run in either unprivileged or restricted-privileged levels. This approach ensures that

Table 3.1 uSFI Privilege Levels and Allowed Access. The table shows the three privilege levels in a uSFI-enabled system and the accesses allowed at each privilege level. The uSFI runtime code is the only software that runs in privileged level. Modules can only have either unprivileged or restricted-privileged privilege levels.

Access Type	Privilege Levels		
	Privileged	Unprivileged	Restricted-privileged
Module's memory	Accessible	Accessible	Accessible
Other modules' private memory	Accessible	Inaccessible	Inaccessible
System control block	Accessible	Inaccessible	Inaccessible
System instructions	Accessible	Inaccessible	Accessible

even if a privileged code such as an RTOS kernel is compromised, other modules' memory is safe from being leaked or altered by privileged malicious code. Table 3.1 shows what accesses are allowed under each privilege level. An unprivileged module only has access to its own memory and other modules' public data. A restricted-privileged module has additional access to system instructions, but it can not access system control resources such as MPU configuration registers and other modules' private memory.

3.3.2 Memory Isolation

uSFI protects module memory from compromised or malicious modules. To achieve this, it uses a Memory Protection Unit (MPU) hardware available in many embedded processors. The MPU divides the memory map into regions and defines access permissions and memory attributes for each region. In a uSFI-enabled system, the memory map is divided into two parts: uSFI memory and module memory. uSFI memory is a small portion of the memory map used by the uSFI runtime. This portion of memory is inaccessible by any of the modules. The rest of memory (module memory) is freely accessible by the corresponding module code. Each module memory region is divided into code, stack, read-only data, private and

Table 3.2 Access Permission Configuration for Module Memory Regions. The table shows MPU access permission configuration for an active module. The module region is configured to be accessible by both privileged (the uSFI runtime) and the currently active unprivileged code. Data and peripheral regions are non-executable (NX). Other modules' regions are left unconfigured, which makes them accessible by only the uSFI runtime.

Module Memory Region	Unprivileged/Privileged Permissions
Code	RO, X
Read-only Data	RO, NX
Stack	RW, NX
Data	RW, NX
Peripherals	RW, NX

public data (bss, data and heap), and peripheral regions.

In a uSFI enabled system, only a single module is active at a time. Inter-module calls are managed by the uSFI runtime. On the invocation of a new module's function, the MPU configuration is changed to reflect the access permissions of the new module. Table 3.2 shows the access permission configuration for each region of the active module. As shown in the table, each region has distinct permission requirements. Data is not executable, therefore all data regions (*i.e.*, read-only data, stack and data) are configured as non-executable (NX). The module regions are configured to be accessible by privileged and unprivileged code, *i.e.*, the active module and the uSFI runtime have access to these regions. The active module only has access to the configured regions. Other modules' regions (unconfigured regions) are made inaccessible to non-privileged code by use of the MPU hardware.

3.3.3 uSFI Compiler and Verifier

uSFI is composed of a uSFI compiler with a verifier and a uSFI runtime. The uSFI compiler generates a binary that conforms to the constraints discussed below. Figure 3.2 shows the steps of generating a binary in a uSFI-enabled system. A programmer specifies module configurations for each module through a uSFI API. The configurations include the module

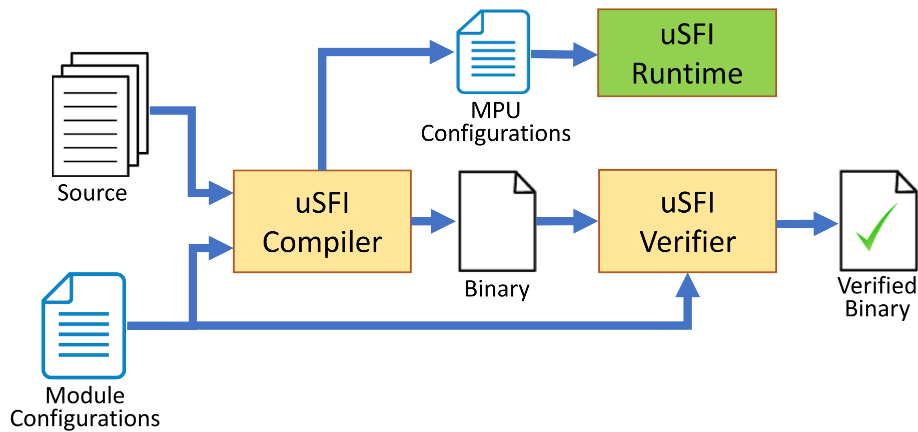


Figure 3.2 uSFI Compiler and Verifier. The figure shows the process of generating a binary and MPU configurations in a uSFI-enabled system. Module configurations are provided for each module through the uSFI API and include the module number, the privilege level of the module and the peripheral access permissions of the module. The uSFI compiler uses the configuration information to allocate memory to modules in module memory. The compiler also generates MPU register configurations to be used by the uSFI runtime. A uSFI verifier ensures that the compiler-generated binary satisfies the restrictions set on module instructions.

---

number, privilege level, peripheral access permissions, and entry function. Based on this information, the compiler selects module memory region sizes and allocates regions to modules. The compiler also generates MPU configurations to be used by the uSFI runtime.

After compilation, a static verifier ensures that the generated binary satisfies restrictions set by uSFI. The restrictions depend on the privilege level of the module and are listed below:

1) Modules can only issue supervisor calls that are assigned to them: Modules issue supervisor calls when they want to make cross-module procedure calls. To keep cross-module call overhead low, the uSFI runtime keeps the amount of checks required during module transition to the minimum. Modules are identified by a unique module number. When a module wants to call a function in a different module, it issues a supervisor call with the callee's module number as an argument. The runtime identifies the module to switch to using this number. However, it doesn't check the source of the call. It is up to the uSFI verifier to make sure that modules issue only allowed supervisor calls (*i.e.*, supervisor calls with the right module numbers).

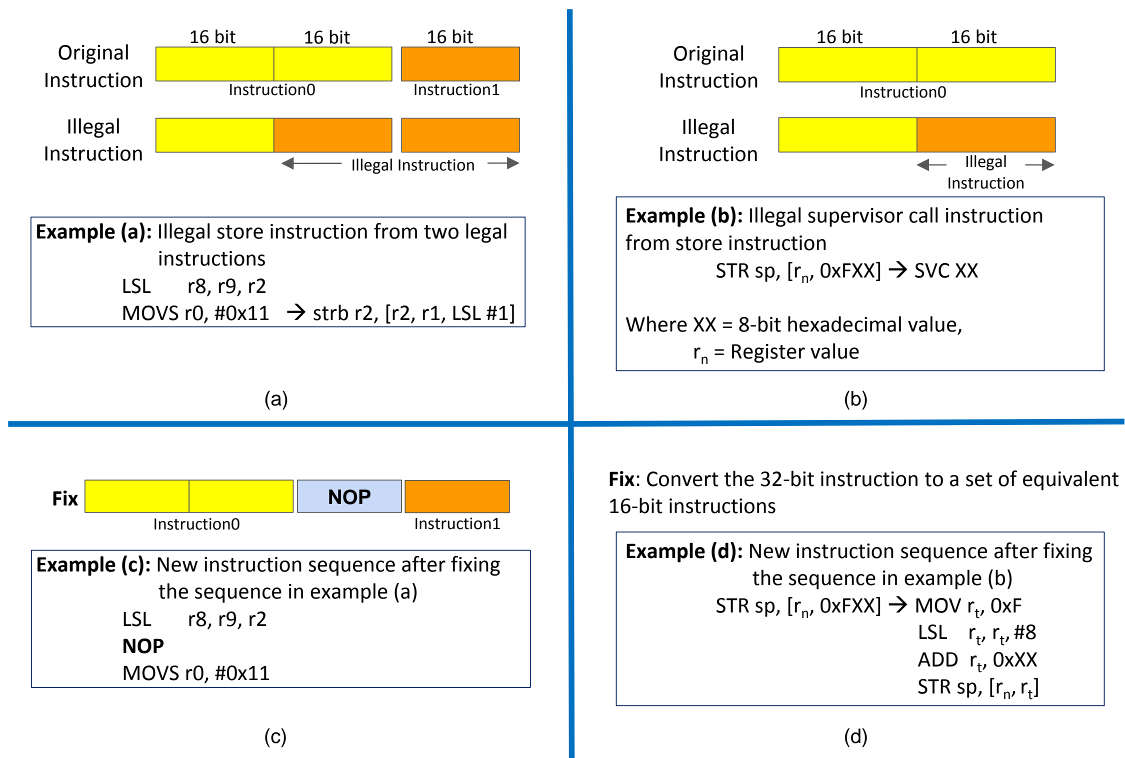


Figure 3.3 Potential Illegal Instructions and How to Fix Them. The figure shows how instructions that violate uSFI restrictions can be formed, and how the uSFI compiler fixes them by taking the ARMv7-M architecture as an example. In (a) a 32-bit illegal instruction is formed by jumping into the middle of a 32-bit instruction and combining it with the next 16-bit instruction. Example (a) shows how this can be used to execute an illegal store operation in a restricted-privileged module. To fix this, the uSFI compiler simply inserts a *NOP* instruction between the two instructions, as shown in (c). (b) shows how a 16-bit instruction is formed by jumping into the middle of a 32-bit instruction. This can be used to execute an arbitrary supervisor call as shown in example (b). The uSFI compiler fixes this by replacing the 32-bit instruction into 16-bit instruction sequence, as shown in (d).

---

2) Privileged modules can access memory using only unprivileged (and thus MPU checked) memory access instructions: In a uSFI-enabled system all code except the uSFI runtime has restricted access to memory. To enforce this, modules with a restricted-privileged level can not use privileged load and store instructions.

In a uSFI-enabled system the uSFI compiler is not part of the trusted computing base. The compiler is expected to generates code that satisfies the above restrictions by ensuring that all code is discoverable at compile time. However, the correctness of the code doesn't solely rely on the rather large compiler. Instead, compiler generated code has to be vetted by

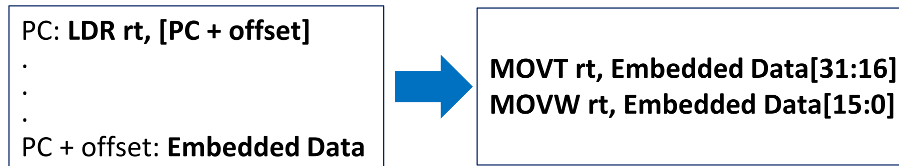


Figure 3.4 PC-relative Load Instruction and its Conversion to Safe Move InstructionsThe figure shows how the uSFI compiler removes data embedded in the code segment by taking the ARMv7-M architecture as an example. In the original code a PC-relative load instruction loads data embedded in the code segment. This code is unsafe since illegal instructions can be formed by jumping into the embedded data. The uSFI compiler removes the embedded data by replacing the load instruction with two 16-bit immediate move instructions.

---

the trusted verifier before it is ready for execution. The vetting process ensures that all code is discoverable at compile time (*i.e.*, it is impossible to form new unexpected instructions at runtime).

In modern architectures, it is possible to form new instructions at runtime that are not observed at compile time through code gadgets that jump into the middle of code and data. For example, in the ARMv7-M architecture this can happen in two ways. First, due to the variable instruction encoding in the ARMv7-M architecture, it is possible to jump into the middle of an instruction and form new instructions. ARMv7-M uses the Thumb-2 instruction set which supports both 32-bit and 16-bit instructions [69]. Instructions are stored half-word (16 bits) aligned and 16-bit and 32-bit instructions can be intermixed freely. Therefore it is possible to form illegal instructions at runtime by: 1) executing two 16-bit instructions as a single 32-bit instruction, or 2) jumping into the middle of a 32-bit instruction and executing the lower half of the instruction.

Figure 3.3 illustrates this potential vulnerability, and our remedy to prevent it from creating illegal code sequences. In Figure 3.3 (a) an example is given on how an illegal store instruction can be constructed by jumping into the middle of a 32-bit instruction and combining it with the next 16-bit instruction. Assuming the code resides inside a module with a restricted-privilege level, this violates the second restriction stated above. Figure 3.3 (b) provides an example that shows how an arbitrary system call can be formed by jumping into the middle of a legal 32-bit instruction. This violates the first restriction stated above.

Figure 3.3 (c) and (d) show how the uSFI compiler fixes these potential violations. To fix scenario (a), a *NOP* instruction is inserted between the two 16-bit instructions. The scenario in (b) is fixed by replacing the 32-bit instruction with an equivalent 16-bit instruction sequence.

Another avenue to generate illegal instructions is through data embedded in code memory. This occurs when using PC-relative addressing to access data. In PC-relative addressing, the data to be loaded is located in the code region at a fixed offset from the program counter. With data embedded in the executable code region, it is possible to form illegal instructions by jumping into the code-segment embedded data. The uSFI compiler deals with this potential vulnerability by excising all data from the code segment. With our baseline compiler (LLVM), code-segment embedded data is limited to immediate values for register loads. The compiler deals with potential violations by simply replacing the instructions with other safe instructions. For example, in the ARMv7-M architecture PC-relative load instructions can be replaced by two immediate move instructions, as shown in Figure 3.4. The replacement does not incur any performance overhead as the two move instructions take the same amount of time to execute as a single load instruction

Algorithm 3.2 Algorithm for uSFI Verifier

```
 1: procedure VERIFYBINARY(Binary)
 2:     size_of_binary ← sizeof(Binary)                ▷ Binary size in bytes
 3:     IP ← 0                                          ▷ Byte index
 4:     while IP < (size_of_binary − 2) do
 5:         Inst16 ← Binary[IP : IP+2]                  ▷ 16-bit instruction
 6:         Inst32 ← Binary[IP : IP+4]                  ▷ 32-bit instruction
 7:         if disallowed_instruction(Inst16) then
 8:             offending_IP ← IP
 9:             offending_inst ← Inst16
10:             return false
11:         if disallowed_instruction(Inst32) then
12:             offending_IP ← IP
13:             offending_inst ← Inst32
14:             return false
15:         IP ← IP + 2
16:     return true
```

Algorithm 3.2 shows the pseudo-code for the uSFI verifier. The algorithm is specific to the ARMv7-M architecture. The verifier scans through the compiled binary and checks all possible 16-bit and 32-bit instructions for potential code violations. The check takes $O(2n)$ time, where n is the number of half-words (16 bits) within the code. Once a binary is verified, it is signed to prevent potential tampering. We assume a bootloader verifies the binary signature on system startup.

3.3.4 uSFI Runtime

The second component of uSFI, the uSFI runtime, manages modules at runtime. The responsibilities of the runtime include handling the switch between modules and handling interrupts/exceptions. The runtime keeps a list of modules and their configurations. The configurations include the value of the stack pointer for the module, MPU configurations, and privilege levels of the module. The runtime also keeps a list of exported functions and their entry points for each module. These are functions that can be called from within other modules.

Inter-module Function Calls One of the tasks of the uSFI-runtime is to handle the switch between modules. A switch between modules is required when a module wants to call an exported function in another module. At compilation, the uSFI compiler installs *gateway functions* in each module to facilitate module switches. The gateway function is an entry point to a module. Modules interact with the runtime through supervisor calls. A supervisor call takes an argument that indicates what the caller is requesting. When a module is created it is assigned a module number. During cross-module function calls, the module number is used as an argument to a supervisor call to indicate the callee's module.

Figure 3.5 shows the steps involved in inter-module function calls. In the figure function *foo_A* in module A wants to call the exported function *bar_B* in module B. In (1) Module A passes a pointer to function *bar_B* in register *rx*. Then it issues a supervisor call, with module

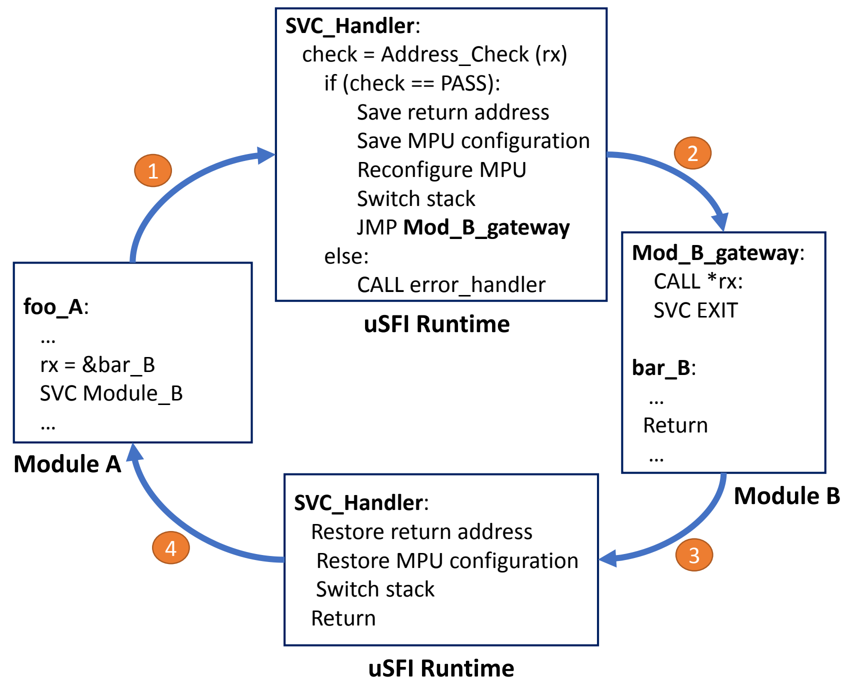


Figure 3.5 Inter-module Function Call. The figure shows the steps involved in calling functions across modules. In the figure, module A wants to call the function *bar_B* in module B. In (1) Module A issues a supervisor call with SVC number of module B as an input to the uSFI runtime after passing a pointer to function *bar_B* in register *rx*. The uSFI runtime verifies the function pointer. If the check passes, the runtime saves the return address and the MPU configurations of module A and changes the MPU configuration to enable memory regions of module B. Then the runtime switches the stack and jumps to the gateway function in module B (2). At the end of execution of function *bar_B*, module B issues an exit supervisor call to the uSFI runtime (3). The uSFI runtime restores the stack and MPU configurations of Module A and transfers control back to module A (4).

---

number of module B as an argument, to the uSFI runtime. (Note that from a programmer's perspective this is a regular function call; the supervisor call instruction and the assignment to register *rx* are automatically inserted by the uSFI compiler). The uSFI runtime verifies that the function pointer points to a function exported by module B. If the check passes, the runtime saves the return address and the MPU configurations of module A. Then, after changing the MPU configuration to enable memory regions of module B and switching the stack, control is transferred to the gateway function in module B (2). At the end of execution of function *bar_B*, module B issues an exit supervisor call to the uSFI runtime (3). Finally, the uSFI runtime restores the stack and MPU configurations of module A and transfers control back to module A (4).

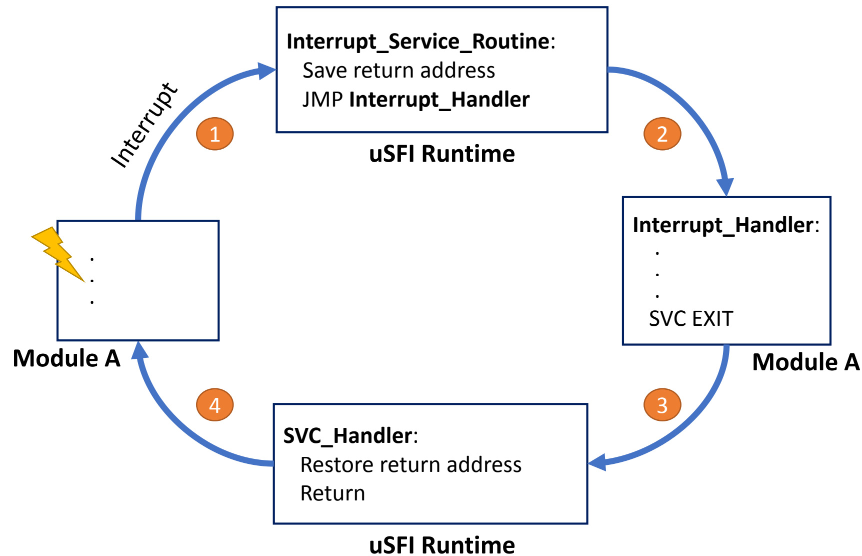


Figure 3.6 uSFI Interrupt Handling. The figure shows the interrupt handling process in a uSFI-enabled system. When an interrupt occurs, control is transfered to the interrupt service routine in the uSFI runtime. The runtime saves the return address for the interrupt and invokes an interrupt handler routine assigned by a module. Normally interrupt service routines run in privileged mode. Therefore, the runtime changes the privilege level before the module routine is invoked. Once the module interrupt handler finishes execution, it issues a supervisor call to the runtime, which then transfers control back to the module instruction where the interrupt occurred.

Interrupt/Exception Handling Interrupt handling is performed by the uSFI runtime. The uSFI API allows modules to assign their own interrupt handler routines to certain events. Figure 3.6 shows the interrupt handling process in a uSFI-enabled system. In the figure, an interrupt occurs while module A is executing. As a result, control is transfered to an interrupt service routine in the uSFI runtime. This routine sets the right privileges, saves the return address and calls an interrupt handler routine assigned by the module. All module interrupt service routines end with a supervisor call instruction. At the end of a module interrupt service routine, an exit supervisor call is issued to the uSFI runtime. In the supervisor call handler, the uSFI runtime restores the return address to module A and transfers control back to module A.

3.4 uSFI Implementation

In this section we discuss our implementation of the uSFI architecture. We implemented uSFI for the ARM Cortex-M based microcontrollers that use the ARMv7-M architecture. The ARMv7-M architecture is a widely used architecture in embedded processors. Although the discussion in this section is specific to ARMv7-M architecture, it also applies to ARMv7-R and the new ARMv8-M/R architectures.

3.4.1 MPU Configuration

For our implementation, we used Cortex-M4 based microcontrollers, although the techniques described here work for all ARM Cortex-M and Cortex-R processors. The MPUs in these microcontrollers allow configuring up to eight memory regions (region 0 to region 7). Region 0 to region 5 are used for code, read-only data, stack, private data (bss, data and heap), and public data, respectively. The remaining three regions are used for peripheral access control. Note that we don't need to explicitly configure any region for the uSFI runtime or inactive modules since any region not included in the configuration is treated as a *background* region and can only be accessed by privileged code (*i.e.*, uSFI runtime).

Embedded devices typically include multiple peripherals. uSFI provides fine-grained peripheral access control by granting modules access to only the peripherals they need. Peripherals in microcontrollers are typically memory-mapped, therefore peripheral access can be controlled by using the MPU. In microcontrollers with eight MPU regions, three of the regions are used to provide peripheral access control. In this case, the peripheral memory region is divided into three regions. Cortex-M4 MPUs allow further subdividing of each region into eight equal-sized sub-regions. This approach allows up to 24 distinct peripheral regions that can be enabled or disabled to provide fine-grained peripheral access control.

Two MPU registers need to be configured for each module region. These are the *MPU Region Base Address Register (MPU_RBAR)* and the *MPU Region Attribute and Size Reg-*

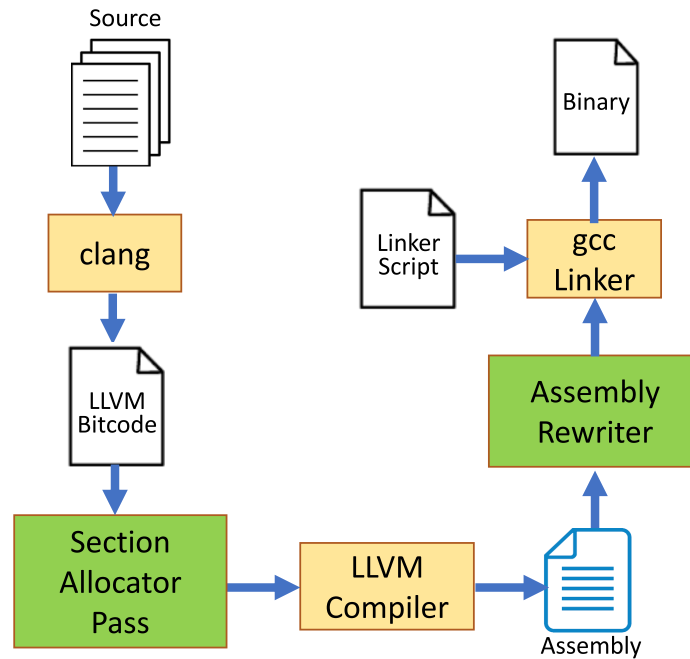


Figure 3.7 uSFI Compiler Implementation. The uSFI compiler is implemented as an LLVM pass and an assembly rewriter. The LLVM pass assigns code and data sections to modules based on configurations provided by a programmer. In addition to this, the pass installs gateway functions in each module to facilitate cross-module function calls. The assembly rewriter converts cross-module function call instructions to supervisor call instructions, and makes sure the restrictions listed in Section 3.3 are satisfied by rewriting potentially illegal instructions and removing embedded data in code sections.

---

ister (MPU_RASR). MPU_RBAR defines a region's start address. MPU_RASR defines a region's size and memory attributes, and enables the region and its sub-regions. MPU register configurations for module stack and peripheral regions are obtained at compile time. Start address and sizes of code and data regions are not known until link time, and therefore MPU register configurations for module code and data regions are resolved at link time.

3.4.2 uSFI Compiler and Verifier

The uSFI compiler is implemented as an LLVM [50] compiler pass and an assembly rewriter. Figure 3.7 shows the compilation process. The LLVM pass identifies module functions and assigns them code sections in such a way that all functions of a module belong to the same code section. This ensures that a module's code occupies contiguous code memory. Module

data (read-only, initialized and uninitialized data) is also assigned sections in data memory. In addition to assigning sections, the LLVM pass also installs a gateway function in each module as shown in Figure 3.5.

Assembly rewriting is used to replace calls to exported functions with supervisor call (SVC) instructions that invoke the uSFI runtime. Register *r4* is used to pass a pointer to the exported function as shown in Figure 3.5. Assembly rewriting is also used to replace all module load and store instructions with unprivileged load and store instructions. Finally, the modified assembly is linked using the gcc linker (ld) with a linker script describing how code and data sections are positioned in physical memory. Module code and data are assigned contiguous regions in memory. uSFI runtime code and data is placed at a fixed, known address in physical memory. The start addresses and sizes of each regions, which are used by the uSFI runtime to configure the MPU, are obtained from the linker.

We implemented a verifier using the radare2 reverse-engineering framework [70]. We tested the verifier on unmodified binaries (compiled with the gcc compiler) from the MiBench benchmark suite [71], and unsurprisingly there were illegal supervisor call instructions. These instructions were found embedded in the code-segment data as shown in Figure 3.3. We found such illegal instructions in four of the benchmarks (qsort, basicmath, rijndael and mbedtls). Embedded data was removed and PC-relative load instructions were replaced by immediate move instructions to ensure that these illegal sequences could not be invoked.

3.4.3 uSFI Runtime

The uSFI runtime is a small code written in C and assembly with the bulk of the code implementing a supervisor call (SVC) handler. It has a total size of less than 150 lines of C and assembly statements. The runtime keeps a list of module MPU register configurations and exported functions with their entry points for each module. It also has a stack that it uses to save and restore MPU register configurations on module switches. The runtime code

and data memory can not be accessed by any module. Furthermore, the MPU configuration registers, and the Nested Vectored Interrupt Controller (NVIC) can only be accessed by the uSFI runtime. This restriction is achieved by making the uSFI runtime code the only privileged software component in the system.

Modules can call exported functions in other modules. During cross-module function calls, function arguments are passed through registers, and when necessary, through the stack. ARM uses registers *r0* to *r3* to pass function arguments; therefore for most cases passing arguments through registers is sufficient. In the rare case of more than four function arguments, the stack is used to pass the additional arguments. In this case, uSFI copies the additional arguments to the stack of the callee's module. In addition to this, uSFI clears registers during module switches to prevent data leaks. Each module includes a public data region that is accessible by all modules. This region is used to pass larger data to other modules.

3.5 Experimental Evaluation

We evaluated uSFI using representative embedded benchmarks. We used two development boards for our evaluations: STMicroelectronics's NUCLEO-F446RE development board [72], and NXP's FRDM-K64F development board [73]. The NUCLEO-F446RE board uses a microcontroller with an ARM Cortex-M4 processor, 512KB flash memory, and 128KB RAM. The FRDM-K64F board uses a microcontroller with an ARM Cortex-M4 processor, 1MB flash memory, and 256KB RAM. During our evaluation, we evaluated code size and performance overheads of uSFI, as well as the trusted code size for the original runtime and a hardened version with many additional runtime checks.

3.5.1 Code Size Overhead

We evaluated the overhead in module code size using benchmarks from the MiBench embedded benchmark suite [71], mbed TLS library [74] and the FreeRTOS kernel [75], a widely used embedded RTOS. There are two sources of code size overhead in uSFI. First, there is minor overhead when replacing PC-relative load and arithmetic operations in module code. PC-relative loads (16-bit instructions) are replaced with two immediate move instructions (32-bit each) as shown in Figure 3.4. 32-bit immediate arithmetic operations are replaced with immediate move instructions followed by register arithmetic operations. The other source of code size overhead is when replacing ”ordinary” load and store instructions with unprivileged load and store instructions in restricted-privileged modules.”Ordinary” memory access instructions are typically 16-bit instructions in the ARMv7-M architecture, while unprivileged memory access instructions are 32-bit instructions.

Table 3.3 shows the code size overhead for benchmarks from MiBench, mbed TLS library, and the FreeRTOS kernel. The *mbedtls* benchmark tests all the cryptographic operations in the mbed TLS library. Except for FreeRTOS, all benchmarks are running as an unprivileged module (*i.e.*, the benchmarks use ”ordinary” load and store instructions). The FreeRTOS kernel is used to evaluate the code size overhead of using unprivileged load and store instructions. Normally the kernel runs in privileged level, and therefore it has access to the entire system memory. But in a uSFI-enabled system, it runs in restricted-privileged level (*i.e.*, it doesn’t have access to task memory). The last row in Table 3.3 shows the code size overhead of sandboxing the FreeRTOS kernel.

In the table *rijndael* has a relatively higher code size overhead among the unprivileged benchmarks. This is due to the large number of PC-relative loads of the same pointers to s-box tables. FreeRTOS has a relatively larger code size overhead since unprivileged load and store instructions are twice as large as the ordinary memory access instructions. Overall, compared to the total size of the flash memory in the devices, the additional code size is small.

Table 3.3 Code Size Overhead. The table shows the code size overhead of replacing PC-relative load and arithmetic operations to remove data embedded in code memory. Here each benchmark except FreeRTOS is assumed to be running as an unprivileged module. FreeRTOS is running as a restricted-privileged module, *i.e.*, in addition to not having access to PC-relative load and arithmetic operations, it uses unprivileged memory access instructions.

Benchmark	Original Code Size (Bytes)	Additional Code Size (Bytes)	% Overhead
dijkstra	11388	576	5.1
susan	51092	406	0.8
basicmath	22880	806	3.5
bitcount	9248	280	3.0
qsort	18572	744	4.0
stringsearch	17484	130	0.7
rijndael	41904	3224	7.7
sha	8676	232	2.7
blowfish	16512	560	3.4
FFT	18008	410	2.3
CRC32	7388	228	3.1
mbedtls	362736	2682	0.7
FreeRTOS	**45360**	**4272**	**9.6**

The results shown in Table 3.3 are obtained by naively removing all embedded data from code memory. Although the overhead is small, it can be further reduced to a negligible size by selectively removing embedded data, *i.e.*, remove only data that can potentially be used to form illegal instructions as discussed in Section 3.3. Finally, it is important to note that there is no performance overhead inside the sandboxes, since instructions (in particular loads/stores and indirect jumps) are not instrumented in any fashion.

3.5.2 Performance Overhead

We also measured performance overhead of uSFI using other highly detailed real-world applications. To measure execution cycles, we used the Data Watchpoint and Trace Unit (DWT) facility available on ARM Cortex-M4 processors [76]. DWT provides clock cycle

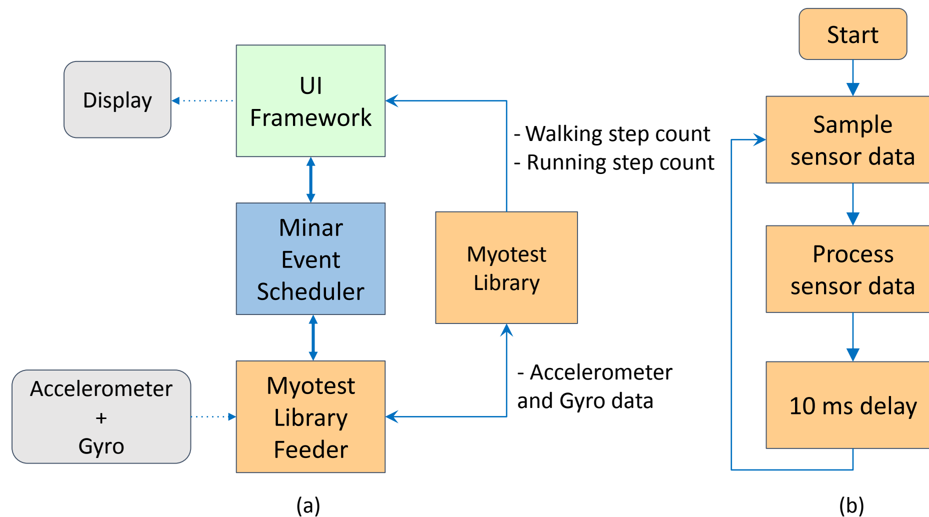


Figure 3.8 ARM Wearable Reference Design Step Analysis Application. The figure shows a block diagram and flow chart of a step analysis application used to evaluate performance overhead of uSFI. The application periodically samples sensor data, calculates walking and running steps and displays the result on a display. In (a) the different modules of the application are shown. The *Minar Event Scheduler* is a scheduler in ARM's mbedOS. *Myotest Library Feeder* and *Myotest Library* are step analysis libraries from Myotest. The *UI Framework* is a user interface library.

---

measurements among other things.

The performance overhead in uSFI comes from module switches during cross-module function calls as shown in Figure 3.5. Overall, it takes 210 cycles to call a function in a different module and 150 cycles to return from the call. To evaluate the effect of module switching on applications' overall performance, we used two applications: a step analysis test application from ARM's Wearable Reference Design (WRD) [77] and an HTTPS file Download application [78].

3.5.3 Case Study 1: Step Analysis

The step analysis application periodically samples sensor data from accelerometer and gyroscope sensors, and calculates walking and running steps. The result is displayed on a matrix LCD. Figure 3.8 shows a block diagram of the application. The application has three major components. The *minar event scheduler* is a non-preemptive event scheduler from mbedOS 3. The *myotest library* along with *myotest library feeder* is a step analysis library

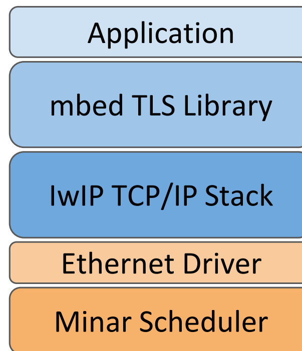


Figure 3.9 Software Modules for HTTPS File Download Application. The figure shows the various software components of the second test case application. Here each component is running in its individual sandbox.

---

from Myotest [79]. The *UI framework* provides an interface to the matrix LCD display. We evaluated the application using the NUCLEO-F446RE development board operating at 120MHz. As a sensor input we used a dataset of 14,255 samples of sensor data provided by Myotest. The output from the application is captured through one of the UART ports on the board.

Sandboxing can be applied at different levels for different components of an application. We measured the performance overhead of sandboxing for two cases. In the first case only the myotest library (with the feeder) was sandboxed. In the second case two sandboxes were used; one for the myotest library, and another for the UI framework. Table 3.4 shows the results for the two cases. The table compares the two results with a baseline implementation where no application components are sandboxed. As can be observed from the table, the overhead of sandboxing the modules is small. This can be attributed to the low sampling rate of the sensors (sampling is done every 10ms) as compared to the processing speed of the processor. This is typical in many embedded systems applications that utilize sensors.

3.5.4 Case Study 2: HTTPS File Download

The second application we tested is an example application for the mbed TLS library. The application downloads a file from an HTTPS server and looks for a specific string in that file. The application runs on ARM's mbedOS embedded operating system. Figure 3.9 shows the

Table 3.4 Execution Cycles for the Step Analysis Application. The table shows execution cycles for the step analysis application for three cases. In the baseline application all components of the application execute in the same security domain. In uSFI 1 only the Myotest library is sandboxed, while in the case of uSFI 2 both the Myotest library and the UI framework are sandboxed.

	Baseline	**uSFI 1**	**uSFI 2**
Total Number of Module Switches	0	42765	43778
Additional Clock Cycles	0	15395400	15760080
Total Clock Cycles	1434843597	1448689953	1449054633
% Overhead	0	1.07	1.10

different software components of the application. Each software component is contained in its own sandbox. We used the NUCLEO-F446RE development board operating at 120MHz for the test.

The baseline application (without uSFI) takes on average 3.2 seconds to execute. With uSFI enabled, an average of 3276 module switches were recorded. Each module switch takes 3us at 120MHz clock frequency. This results in an average overhead of only 0.31% overhead. Most of the overhead comes from the Ethernet driver which is invoked every 1ms.

3.5.5 Trusted Code Size

We also measured the code size of the trusted uSFI runtime. The runtime has a code size of only 1.2kB, resulting in a very small attack surface. To make the runtime even more resilient to potential attacks, we then hardened it by manually inserting bounds checking instructions before critical operations. The code size grew to only 1.4kB.

To see the effect of the hardened runtime on the performance of applications, we run the step analysis application with the hardened runtime. The maximum runtime overhead increased only very slightly to a modest 1.3%.

3.6 Related Work

In this section we review previous works, focusing on embedded systems security, particularly code and data isolation.

3.6.1 Software-based Fault Isolation

Wahbe et al. [80] introduced software-based fault isolation as a more efficient way of providing isolation between tightly-coupled software modules within the same address space. This is a cheaper alternative to placing modules in separate address spaces and using Remote Procedure Calls (RPC) to call into each other. To lock down modules within their fault domains, SFI inserts address checking instructions before every unsafe instruction. Since then several works have proposed using variants of SFI for different applications and platforms [81, 82, 83]. Native Client (NaCl) [81] uses SFI to allow running native C/C++ code within a web browser. NaCl sets constraints on untrusted binaries and uses a validator to make sure these constraints are met. A follow-up work [82] extends support of NaCl to x86-64 and ARM platforms. XFI [83] combines SFI and Control-Flow Integrity (CFI) techniques to protect host environments (*e.g.*, kernel, web browser) from corruption by modules that operate within the same address space as the host environment. Example modules include drivers and DLLs.

ARMlock [84] proposes efficient fault isolation for the ARM architecture by using memory domain support available on ARM mobile processors. In a similar fashion, our work uses hardware support to reduce the performance overhead of sandboxing. However, unlike uSFI, ARMlock targets mobile processors and assumes the kernel is trusted.

3.6.2 Sandboxing in Embedded Devices

Several works have proposed hardware and software techniques to provide isolation in embedded devices. TrustLite [32] and Tytan [85] propose a hardware extension, Execution-

Aware Memory Protection Unit (EA-MPU), to provide isolation of trusted modules from untrusted code including untrusted OS. Similarly uSFI assumes the OS kernel is untrusted, but doesn't require any hardware changes. Furthermore, Trustlite uses static hardware configuration table to configure the EA-MPU , which means the area overhead grows quickly with the number protected modules. This limits the number of protected modules that can be supported. By allowing a small trusted runtime to configure the MPU, uSFI can support unlimited number of software modules.

Other line of research has looked at software-only solutions to isolate software modules. ARMor [33] uses SFI to sandbox non-critical code. It uses binary rewriting to put checks before store operations identified as being potentially unsafe. At runtime, It uses a separate *control stack* to protect return addresses. Similarly, [34] and [35] use a separate stack to protect return addresses. Other indirect control flow instructions are validated by runtime checks. To reduce the overhead of the runtime checks, these techniques only provide write protection. In addition to this, the added memory guard instructions result in large code size and performance overhead. uSFI provides both memory read and write protections with negligible inner sandbox performance overhead.

ARM mbed uVisor [86] is a software hypervisor that creates independent secure domains called *boxes*. Like uSFI, uVisor uses the MPU to provide isolation and access control to peripherals. However, there are some important differences between uSFI and uVisor. First, in uVisor MPU configurations are tied to process switches, *i.e.*, MPU configuration changes are done at process context switches. This makes isolation between *tightly-coupled* software modules expensive. The only way to provide isolation between tightly-coupled components with uVisor is to put the two components in separate boxes, and use synchronous remote procedure calls (RPCs) to call into each other's boxes. But RPCs require process switches, which means that many cycles are wasted while waiting for the switch. This makes uVisor unsuitable to provide isolation between tightly-coupled software modules. On the other hand, in a multi-process system with uSFI, module switches and process switches are separate,

allowing inter-module calls to be serviced immediately.

Another important difference between uSFI and uVisor is that an RTOS integrated with uVisor runs with the same privilege as uVisor. That means the RTOS has access to MPU configuration registers and process memory. On the other hand, in a uSFI-enabled system only the uSFI runtime has access to MPU configuration registers. Privileged code such as an RTOS is sandboxed using non-privileged memory access instructions.

Another protection recently added to embedded devices is ARM TrustZone [87]. TrustZone allows partitioning software into secure and normal worlds and provides isolation between the two. Software in the secure world can access memories in both secure and normal worlds, while normal software can only access normal world (non-secure) memories. TrustZone provides new instructions that facilitate switching between the secure and non-secure states.

While both uSFI and ARM TrustZone provide software isolation, there is an important difference between the two. TrustZone has only two security domains (secure and non-secure), and therefore it can not provide fine-grained isolation between tightly-coupled modules. On the other hand, uSFI allows as many security domains as are needed. In addition to this, the TrustZone feature is available on the upcoming devices that support the ARMv8-M architecture, while uSFI can also be deployed on older devices as long as they have memory protection units (*e.g.*, devices with ARMv7-M and ARMv7-R architecture).

Some recent works have proposed other mechanisms that enhance the security of embedded devices. Clements et al. [88] proposes privilege overlays to limit the time that a bare metal program executes at the privileged level to only operations that require privileged level. It uses a compiler and manual annotation to identify operations that require privileged level and insert supervisor call instructions to elevate privilege. The work also proposes a modified safe stack to defend against control-flow hijacking attacks. nesCheck [89] uses whole-program static analysis and dynamic checking instrumentation to provide memory safety for programs written in *nesC*, a dialect of the C language optimized for resource

constrained embedded devices. C-FLAT [90] provides remote attestation of an embedded system application's control-flow path using TrustZone as a trust anchor. Sancus [91] proposes a hardware extension to provide remote attestation of code running on embedded platforms that allow mutually distrusting parties to run their software modules on the same node. In Sancus only the hardware is trusted.

3.7 Chapter Summary

In this chapter we presented uSFI, a low-cost code and data isolation mechanism for resource constrained embedded devices. uSFI uses the memory protection unit (MPU) hardware available in many embedded devices along with static software analysis to provide stronger security guarantees at a lower cost than previous work. In a uSFI-enabled system, an application is composed of sandboxed modules. Modules, including privileged modules (*e.g.*, RTOS kernel), are untrusted. Only a static binary verifier and a small runtime are trusted. uSFI doesn't require any hardware changes and incurs only 10% code size overhead and roughly a 1% performance overhead on representative applications.

Chapter 4

Detecting Rowhammer Attacks with Hardware Performance Counters

4.1 Introduction

Errors in DRAM devices have been studied for many years. A significant number of studies have revealed reliability issues in DRAM devices ranging from transient errors due to cosmic rays and alpha particles, to permanent errors caused by manufacturing defects and device wear-out [92, 93, 94, 95]. Though researchers have shown the potential security implications of these types of errors [96, 97], the unpredictability and low frequency of occurrences of these errors have made their exploitations less practical.

Recent studies, however, have revealed that disturbance errors in DDR3 and DDR4 DRAM devices pose a major threat for system security [36, 37, 41]. One form of disturbance errors called *rowhammering* allows the manipulation of data in a DRAM row by repeatedly accessing (or "*hammering*") adjacent rows. Errors caused by rowhammering are highly reproducible, making them an ideal target for exploitations.

Recent work by Google's Project Zero [37] has shown how to leverage rowhammer-induced bit-flips as the basis for security exploits that include malicious code injection and memory privilege escalation. Being an important security concern, industry has attempted to defend against rowhammer attacks. Deployed defenses employ two strategies: (1) doubling the system DRAM refresh rate and (2) restricting access to the *CLFLUSH* instruction that attackers use to bypass the cache to increase memory access frequency (i.e., the rate of

rowhammering).

In this chapter we demonstrate that such defenses are inadequate; We show a rowhammer attack that does not require the *CLFLUSH* instruction. This attack bypasses the cache by manipulating cache replacement state to allow frequent misses out of the last-level cache to DRAM rows of our choosing. We also show from our experiments that, using the *CLFLUSH* instruction, it is possible to generate bit flips even if the DRAM refresh rate is increased to 4x.

To protect existing systems from more advanced rowhammer attacks, we develop a *software-based* defense, ANVIL, which successfully thwarts rowhammer attacks on existing systems. ANVIL detects rowhammer attacks by tracking the locality of DRAM accesses using existing hardware performance counters. Our detector identifies the rows being frequently accessed (i.e., the aggressors), then selectively refreshes the nearby victim rows by reading from them to prevent bit flips. Experiments on the SPEC2006 benchmarks show that ANVIL has less than a 1% false positive rate and an average slowdown of 1%. ANVIL is low-cost and robust, and our experiments make a strong case that it is an effective approach for protecting existing and future systems from even advanced rowhammer attacks.

4.2 Breaking Current Mitigation Techniques

Multiple techniques have been proposed to protect existing systems from DRAM disturbance errors. Currently deployed mitigation techniques include doubling the DRAM refresh rate and disallowing cache flush instruction. In this section we show that these techniques are insufficient to guarantee protection from rowhammer exploits. First we show that a refresh period of 32ms is sufficient time to implement a rowhammer attack. Then we show that it is possible to implement a rowhammer attack without using the CLFLUSH instruction.

Hammer Technique	Minimum Number of DRAM Row Accesses	Time to first bit flip
Single-Sided with CLFLUSH	400K	58 ms
Double-Sided with CLFLUSH	220K	15 ms
Double-Sided without CLFLUSH	220K	45 ms

Table 4.1 Rowhammer Attack Characteristics: The measured performance of three rowhammer techniques, i.e., single and double-sided hammering and with/without CLFLUSH to flush the cache. The experiments are run on a Ubuntu-based Sandy Bridge laptop and a 4GB DDR3 DRAM module. The table gives the total number of DRAM row accesses each attack variant is able to produce in 64ms and the time until the first bit-flip.

4.2.1 Rowhammering under a Double Refresh Rate

After DRAM disturbance errors and their security implications were widely recognized, a number of vendors published BIOS updates that double the rate at which DRAM cells are refreshed [39, 40]. By refreshing the DRAM cells more frequently, it is believed that there is insufficient time to carry out a rowhammer attack. We perform experiments on a commodity platform that show that this belief is indeed false. Even when refresh intervals are reduced to 32ms, it is still possible for a malicious program to cause bit flips by repeatedly accessing two rows adjacent to a victim row using a hammering technique dubbed double-sided rowhammering [37]. Table 4.1 lists our experimental results for three rowhammer attacks. We perform experiments on a system with an Intel core i5-2540M processor (Sandy Bridge) and a 4GB DDR3 DRAM module while running Ubuntu 14.04 LTS. As shown in the results of Table 4.1, it is possible to employ double-sided row hammering using the CLFLUSH instruction to flip bits in only 15ms on our DDR3 module—well below the 32ms window of deployed defenses.

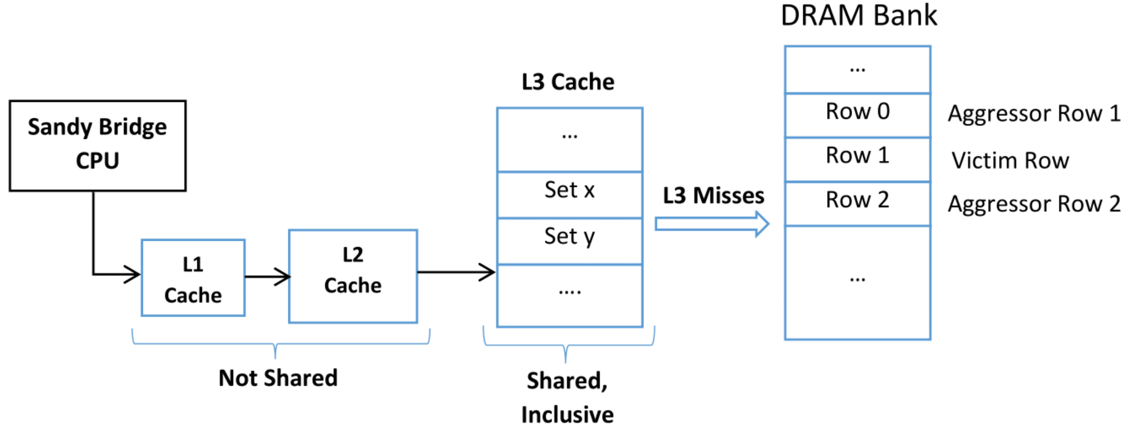
Sequence (a) in Figure 4.1 shows the access sequence used to implement our double-sided rowhammer attack using CLFLUSH instructions. The attack involves three rows: Rows 0 and 2 are the aggressor rows, and Row 1 is the victim row. The aggressor rows are

repeatedly activated to increase the discharge rate in the victim row. The attack works by first accessing an address in Row 0 (i.e., $A0_{(row0)}$); Then an address in Row 2 ($A1_{(row2)}$) is accessed. After each access, a CLFLUSH instruction is used to flush all levels of cache thereby ensuring the next access goes directly to the DRAM. This sequence is repeated N number of times; for our experiments the minimum value of N was equal to 110k to see a bit flip.

Given the results of our experiment, one might suggest further increases in refresh rate. The problem with this approach, in general, is that increasing the refresh rate comes at the cost of increased power and reduced DRAM throughput—as refresh commands compete with software-requested memory accesses. Going from a 64ms refresh period to the 15ms required to protect our DRAM—others may be more sensitive to hammering—requires over a 4x increase in refresh power and throughput overhead. Also, as DRAM continues to move to smaller feature sizes, the vendors will have to lower the refresh rate more to account for increased density (i.e., future DRAM will likely be more susceptible to rowhammering at the cell level).

4.2.2 Rowhammering without the CLFLUSH instruction

Modern processors include multiple levels of cache for faster access of frequently used data. It is common to have three or more levels of cache with the last-level cache capable of storing megabytes of data. In order to repetitively open and close a DRAM row, memory access to that DRAM row must miss on all cache levels and the DRAM row buffer. One way to achieve this is to use cache flushing instructions like CLFLUSH on the x86 architecture. Previous works on exploiting the rowhammer problem all used the CLFLUSH instruction to bypass caches. One counter measure that has been taken to thwart CLFLUSH based attacks is to disallow the CLFLUSH instruction [37]. Such measures thwart rowhammmer attacks based on cache flushing, but we show that it is possible to implement a rowhammer attack without using cache flush instructions.



a) CLFLUSH Attack: $(A0_{(row0)}, CLFLUSH(row0), A1_{(row2)}, CLFLUSH(row2))^{N=110K}$

b) CLFLUSH-free Attack: $(A0_{(row0,\ setx)}, X1_{(setx)}, X2_{(setx)}, ..., X9_{(setx)}, X10_{(setx)},$
$X11_{(setx)}, X1_{(setx)}, X2_{(setx)}, ..., X9_{(setx)}, X12_{(setx)},$
$A1_{(row2,\ sety)}, Y1_{(sety)}, Y2_{(sety)}, ..., Y9_{(sety)}, Y10_{(sety)},$
$Y11_{(sety)}, Y1_{(sety)}, Y2_{(sety)}, ..., Y9_{(sety)}, Y12_{(sety)})^{N=110K}$

Figure 4.1 Memory Access Patterns for CLFLUSH-based & CLFLUSH-free Double-sided Rowhammer Attacks: In (a), a CLFLUSH instruction is used to flush caches after accessing aggressor DRAM rows Row 0 and Row 2. This sequence of operations is repeated N times. In our experiments the minimum value of N observed was 110K. In (b) the CLFLUSH instructions are replaced with sequences of memory accesses that force misses in the L3 cache at addresses that map to aggressor Row 0 and aggressor Row 2. This is done by accessing conflicting data that belong to the same cache set as the aggressor row addresses. *A0:* address in aggressor row 1 and maps to set X; *A1:* address in aggressor row 2 and maps to set Y; *X1, X2, ... X12:* addresses that map to cache set X and evict A0;*Y1, Y2, ... Y12:* addresses that map to cache set Y and evict A1.

Rowhammering in the presence of caches: One way to force a miss from a cache without using the CLFLUSH instruction, is to evict previously accessed data by accessing conflicting data that belongs to the same cache set. To accomplish this, an *eviction set* that contains addresses that belong to the same cache set is created. Then the addresses are accessed one after the other to force eviction of a particular data element from the cache. If the access sequence is cleverly designed to manipulate the cache eviction policy, it is possible to precisely control which addresses hit in the cache and which addresses miss the cache and make it to main memory. This minimizes the delay between subsequent target misses. By repetitively evicting data from a target address and then re-accessing it, corresponding DRAM rows can be activated. Nonetheless, there are significant challenges in

devising efficient address reference streams that can implement a rowhammer attack. First, last-level caches in modern processors have high associativity, usually 8-way to 16-way. Because each way can hold the address of the aggressor row, we must generate at least as many conflicting memory accesses as there are ways. Therefore, many memory accesses are required to evict a cache block, which slows down the hammering process. A second problem is that replacement policies used on real hardware are not true LRU (and vendors usually do *not* publicly disclose them). This means that access patterns that assume true LRU replacement policy do not often result in misses on the required target addresses. Missing on the exact target addresses is important as creating extraneous memory acceses dramatically decreases the rate of hammering.

Demonstration of the attack: In this Section we will describe how we were able to surpass the challenges mentioned above to do CLFLUSH-free rowhammering. For our demonstration we use a processor with an Intel Sandy Bridge micro-architecture. The processor has three levels of cache. The last-level cache is an inclusive, shared, physically indexed 12-way cache. It is an inclusive cache, such that it is enough to evict a word from the last-level cache to bypass the whole cache hierarchy.

One way to create an eviction set is to directly use physical addresses and select memory addresses with the same set-index bits. But since Intel does not publicly disclose physical address to cache set mapping, some reverse engineering is required. Previous work in this area has revealed the mapping for the Intel Haswell microarchitecture [98]. Seaborn [99] discovered that the Sandy Bridge micro-architecture used a slightly modified version of this mapping. In our eviction set we have one address that belongs to a row (which we call an aggressor address). Since our cache is a 12-way cache, we need 13 addresses in the eviction set: 12 conflicting addresses and the aggressor address. We create an eviction set by first picking the aggressor address and then using its physical address to find 12 more addresses with matching cache set mapping. On our Intel Sandy Bridge machine, bits 6 to 16 of the

physical addresses are used to map to last-level cache sets. Furthermore, the last-level cache is organized into *slices* [100], with one slice per processor core. Conflicting addresses will have the same cache slice and cache set-index bits.

The next step is to create an efficient memory access pattern that has a high probability of misses on the aggressor address. Creating such a pattern requires knowing the cache replacement policy of the micro-architecture. Since this is not publicly disclosed, we had to reverse engineer the replacement policy. We did this by generating a high miss-rate pattern that cyclically accesses the 13 addresses, and using performance counters (particularly the last-level cache miss counter) to determine whether each access was a cache hit or a cache miss. Then we correlate the performance counter results with results from different cache replacement policy simulators that we built. Our results show that one of the replacement algorithms Sandy Bridge favors (it uses more than one) is Bit Pseudo-LRU (Bit-PLRU) which is similar to the Not Recently Used (NRU) replacement policy [101]. In Bit-PLRU, each cache line in a set has a single MRU (Most Recently Used) bit. Every time a cache line is accessed, its MRU bit is set. When the last MRU bit is set, the other MRU bits in the set are cleared.

A time efficient access pattern misses the last-level cache only on the aggressor address and one additional conflicting address, and hits on the rest of addresses in the eviction set. This works by always driving the aggressor address to the least recently used position in the replacement state. Sequence (b) in Figure 4.1 outlines the access pattern we used for our CLFLUSH-free double-sided rowhammer attack. This attack is similar to the CLFLUSH-based attack except here the CLFLUSH instructions are replaced with memory accesses that drive the two aggressor DRAM row addresses to the least recently used (LRU) position in the L3 cache and subsequently evict them, thereby ensuring their next access goes to the aggressor DRAM rows. In Figure 4.1b, address $A0_{(row0,setx)}$ belongs to Row 0 in the DRAM, and *SetX* in the L3 cache. Address $A1_{(row2,sety)}$ belongs to Row 2 in the DRAM, and *Sety* in the L3 cache. The two addresses constitute the aggressor addresses. First, data

at address $A0_{(row0,setx)}$ is accessed. Then 10 addresses ($X1_{(setx)}$ to $X10_{(setx)}$) that belong to *Setx* are accessed to put $A0_{(row0,setx)}$ to the LRU position of the L3 cache. Then, when data from address $X11_{(setx)}$ is accessed, data at Address $A0_{(row0,setx)}$ is evicted from the L3 cache. The next 9 accesses ($X1_{(setx)}$ to $X9_{(setx)}$) hit in the L3. Then, after data at address $X12_{(setx)}$ is accessed, address $X11_{(setx)}$ is put to the LRU position and subsequently replaced by data at address $A0_{(row0,setx)}$. This access sequence is repeated N times, with only two addresses ($A0_{(row0,setx)}$ and $X11_{(setx)}$) missing for each iteration. In *sety*, a similar access pattern is used to miss only from addresses $A1_{(row2,sety)}$ and $Y11_{(sety)}$. Using this technique, accesses to Row 0 and Row 2 will always access the DRAM.

Access to the last-level cache on Sandy Bridge takes 26 to 31 cycles [100]. Considering a DRAM access latency of 150 cycles, the access pattern in sequence b) in Figure 4.1 takes an estimated (29*20) + (2*150) = 880 cycles. On our test machine, which runs at a nominal frequency of 2.6GHz, this access pattern takes approximately 338 nanoseconds. This allows up to 190K double-sided hammers with-in a 64ms refresh period. This is enough to produce a flip on our test DRAM module—which only requires 110k accesses to produce a bit flip.

Table 4.1 compares the minimum number of DRAM row accesses and the corresponding time required to produce a bit flip for CLFLUSH-based and CLFLUSH-free attacks for our test DRAM modules. Double-sided, CLFLUSH-based row hammering is the most aggressive of the three. It is also worth noting that a double-sided CLFLUSH-free rowhammering can produce bit flips faster than single-sided CLFLUSH based hammering.

It is interesting to note that if both of the protection mechanisms detailed in this section were used in tandem (i.e., double refresh plus restricted access to CLFLUSH), such a system would still today have a measure of protection against rowhammer attacks, including those detailed in this chapter. As shown in Table 4.1, we are unable to yet rowhammer memory in less than 32ms without use of the CLFLUSH instruction. While we are unaware of any systems that combine these two protection measures, one that did would likely only acquire a temporary measure of protection against novel rowhammer attacks. We continue to optimize

the performance of our CLFLUSH-free attack, and if we are able to reduce its time-to-first bit flip by an additional 13ms, the combined protections will no longer work. Recognizing the tenuous nature of today's rowhammer protections, we feel a better approach to protect systems is to provide in situ mechanisms that detect and subsequently defeat rowhammer attacks.

In summary, current techniques used to protect systems from rowhammer attacks are insufficient. We show that reducing DRAM refresh period to 32ms is not sufficient as faster rowhammer attacks are possible using double-sided rowhammering in as little at 15ms. Moreover, by manipulating the LRU chain of the last-level cache, enough DRAM row activations can be performed in a single refresh cycle to flip bits, without using the CLFLUSH instruction.

4.3 Software-Based Rowhammer Detection and Protection

As Section 4.2 shows, currently deployed rowhammer defenses are insufficient. What is needed is a more robust solution that can detect hammering activity in time to protect any potential victim rows. In this section, we introduce a software technique that uses existing hardware performance counters in commercial processors to detect hammering activity and perform selective refresh on potential victim rows.

4.3.1 Detecting Rowhammer Attacks

Rowhammering relies on repetitively accessing an aggressor DRAM row within a single refresh cycle. We make the observation that this fundamentally requires accesses to the aggressor rows to miss on all cache levels. This reveals two identifying characteristics of rowhammering: *high cache miss rate* and *high temporal locality of DRAM row accesses*. This is in contrast to general memory access patterns where high locality results in high

cache hit rates. As such, it is straightforward to discriminate between rowhammer attacks and non-malcious programs by looking at DRAM access patterns and rate.

Another property of rowhammer attacks is *high bank locality*. DRAM disturbance errors occur due to repeated opening and closing of a row. When a row is accessed, it is opened and its data is transferred to a row-buffer. Subsequent accesses to the same row are served by the row-buffer. In order to close the row, a different row located in the same bank must be accessed. Therefore, a rowhammer attack involves repeatedly accessing at least two rows within the same bank—otherwise the row buffer would prevent the hammering. This bank locality property can be used to differentiate between ”real” row hammering and false positives that are caused by thrashing access patterns observed in some applications.

To minimize the performance impact of rowhammer detection, we propose a two-stage detection mechanism. In the first stage, we monitor the last-level cache miss rate. If this rate is high enough to successfully implement a rowhammer attack, the second stage samples the physical addresses of the memory accesses that miss from the last-level cache. If the samples reveal DRAM row accesses with high temporal locality, then the detector signals this as a potential rowhammer attack. To reduce the possibility of false positives, the detector also verifies that the samples have bank locality. If there is enough bank locality among samples, then a protection phase follows.

4.3.2 Protecting Potential Rowhammer Victims

When the detector identifies potential rowhammering activity, it identifies the potential victim DRAM rows. Victim rows are adjacent to (above and below) identified aggressor rows. To protect the victim rows we refresh them by reading a word from them. Reading from a row opens that row which has the effect of refreshing cells in the row [36]. This approach does not incur significant performance penalties even in the case of false positives.

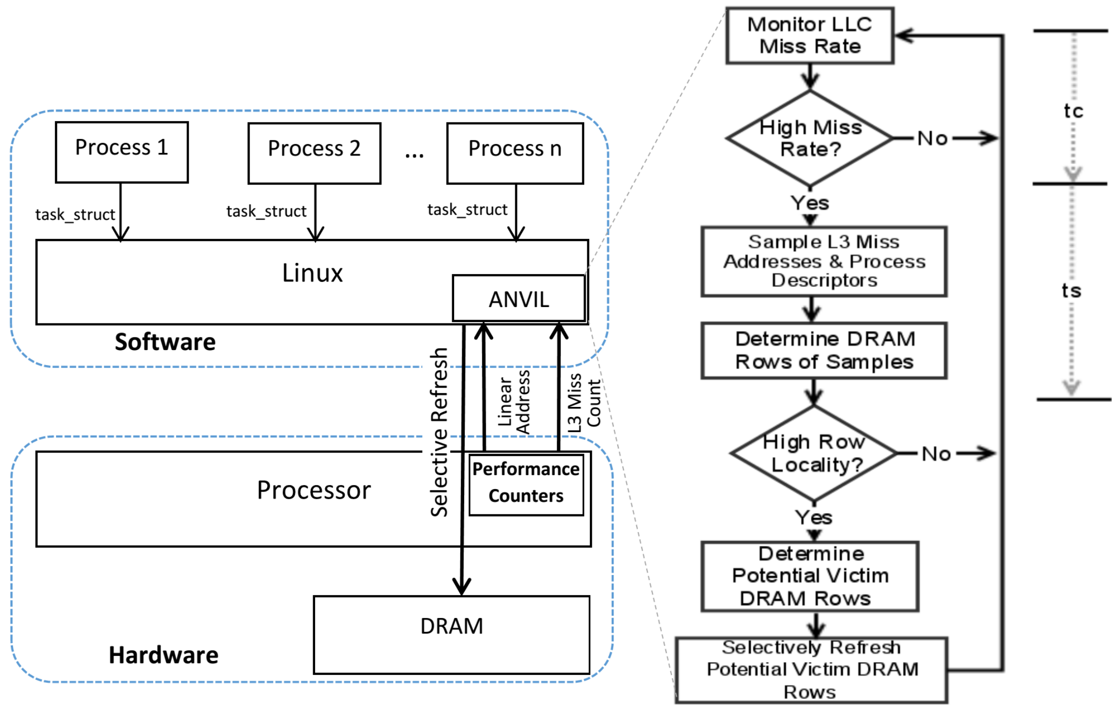


Figure 4.2 Software-Based Rowhammer Attack Detector: ANVIL is a kernel module. It gets last-level cache miss count and memory access samples from hardware performance counters. By combining sampled virtual address information and process descriptor structures, samples of DRAM row accesses are obtained. ANVIL then checks the samples for high locality that suggests potential rowhammer activity. Upon detection of potential rowhammer activity, ANVIL performs selective read operations to refresh victim DRAM rows.

4.3.3 ANVIL: A Linux-Based Rowhammer Protection Mechanism

To demonstrate the protection mechanism, we built ANVIL, a Linux kernel module that prevents all known forms of rowhammer attacks. The module uses hardware performance counters found in modern processors to get memory access information, such as the addresses of loads and stores and the miss rate of the last-level cache. Specifically, we used performance counters found in Intel microprocessors with Sandy Bridge and later microarchitectures. AMD also provides similar capabilities required for our implementation [102]. In this section we provide details of our implementation. We start by reviewing the performance counter features used in our implementation.

Load Latency Performance Monitoring Facility: The Load Latency performance monitoring facility is part of Intel's Performance Event Base Sampling (PEBS) feature. PEBS uses a debug store mechanism and a performance monitoring interrupt to store a set of architectural states [103]. The load latency facility measures latency of a load operation from the load's first dispatch until final data writeback from the memory subsystem. The load operation is sampled probablistically by hardware. If the latency of the sampled load operation exceeds a latency value specified by a dedicated programmable register, the operation is tagged to carry the following information:

- Load data virtual address
- Data source
- Latency value

When the next event categorized as a precise event (e.g. ”load retired”, ”store retired”) occurs, the last update of the load information is written to a PEBS record which then can be read by software. By setting the latency threshold to match last level cache miss latency, it is possible to sample last-level cache misses. The data source information confirms the source of the load operation.

Precise Store Facility : The Precise Store facility complements the Load Latency facility by providing additional information about sampled store operations. When a precise event occurs, hardware samples the virtual address and data source of the next store that retires. Similar to the Load Latency event, data source information can be used to determine the store was a miss.The precise store facility is replaced with the Data Address Profiling facility on Intel's Haswell and later microarchitectures [103]. This facility profiles load and store memory events similar to the other facilities, but has support for more events like DRAM access events. While we could implement ANVIL with either performance counter, we use the Precise Store facility for our implementation since it allows our rowhammer detection mechanism to support older micro-architectures.

In addition to the previously mentioned facilities, we utilize the last-level cache miss

counter to monitor the last-level cache miss rate.

Rowhammer Detection: Figure 4.2 shows the process of detecting a rowhammer activity in ANVIL. In the first stage of the detection phase, the last-level cache miss count event (LONGEST_LAT_CACHE.MISS) is used to measure the last-level cache miss rate. The miss rate is calculated by reading the last-level cache miss count for a time duration of t_c. If this rate is beyond a last-level cache miss threshold (LLC_MISS_THRESHOLD), the second stage of the detector is triggered. The last-level cache miss threshold is set by considering the minimum cache miss rate that is enough to cause bit-flips within a single refresh period. As will be described in the next section we set this value based on our empirical observations. The value can be easily changed to adapt to other systems.

In the second stage, ANVIL samples virtual addresses for a time duration of t_s using Load Latency (MEM_TRANS_RETIRED.LOAD_LATENCY) and Precise Store (MEM_TRANS_RETIRED. PRECISE_STORE) events. The load latency facility allows sampling of loads that have latency beyond a preset clock cycle value. We set the clock cycle value to match last-level cache miss latency so that we only sample loads that miss in the last-level cache. The counter also provides information about the source of the sample, therefore we can ensure the load accessed DRAM. The precise store facility is used to sample stores. It also provides information about the source of a store operation. Which facility to use for sampling is selected based on a count of retired memory load operations that missed from the last-level cache (MEM_LOAD_UOPS_MISC_RETIRED_LLC_MISS) for a time duration of t_c. ANVIL compares this value with the total number of last-level cache misses for that duration. If load operations account for more than 90% of all misses then only loads are sampled. On the other hand, if load operations account for less than 10% of all misses, only stores are sampled. For the remaining cases, both stores and loads are sampled. Load and store sampling rates are adjustable. For our experiments, we used a sampling rate of 5000 samples per second which gives an average of 30 samples for a

sampling duration of 6ms. The second stage also samples the process descriptor (task_struct) of the process that generated the memory access. This structure is used to determine the physical address and DRAM row of the memory access in combination with the sampled virtual address.

At the end of sampling, sampled DRAM row accesses are sorted and the sample distribution is analyzed to identify high DRAM row locality. DRAM row locality is determined by considering the number of samples, the number of last-level cache misses for the sampling duration and the required last-level cache miss rate for a successful rowhammr attack. For each row that has high DRAM locality, a check is made to see if there are other row access samples from the same DRAM bank. If the cumulative of samples of the other row accesses from the same DRAM bank is high enough, then there is a potential rowhammer attack occurring.

Rowhammer Protection: Once ANVIL detects potential hammering activity, we use the physical addresses of the identified aggressor rows to determine potential victim rows. The kernel module was pre-configured using a reverse engineered physical address to DRAM row and bank mapping scheme. We also make the assumption that sequentially numbered rows are physically adjacent. Two potential victim rows are considered for each potential aggressor row: rows that are directly above and below each potential aggressor row (our approach easily extends to N adjacent rows). ANVIL performs a single read operation per victim row to refresh its value. The number of selective read operations performed on a potential victim row is low enough (once every $t_c + t_s$ in the worst case) that it has little effect on performance of non-hammering applications, even if they experience a high incidence of false positive detection. Also, it is not possible for an attacker to use the selective refresh mechanism to rowhammer DRAM rows adjacent to the potential victim row since the selective read rate is well below the minimum access rate for a rowhammer attack. After performing a selective refresh, ANVIL starts the detection process again.

ANVIL Limitations: Although effective, ANVIL's detection mechanism has some limitations. The ANVIL detection mechanism wouldn't be able to detect rowhammer attacks in some scenarios. For example, since ANVIL relies on CPU performance counters, it wouldn't be able to detect attacks that use direct memory access (DMA) [104]. Similarly, trusted execution environments, such as Intel's SGX, do not allow observation of program characteristics using performance counters. Attacks that originate from an SGX enclave [105] would not be detected by ANVIL.

4.4 Experimental Evaluation

In this section we evaluate accuracy and performance of our software detection mechanism. All tests are conducted on a system with an Intel Core i5-2540M processor and Ubuntu 14.04 LTS with Linux kernel version 4.0.0.

4.4.1 Benchmark Applications

We use several benchmark programs for our evaluations. To evaluate rowhammer detection accuracy, we use two rowhammer attacks. The first is a CLFLUSH-based double-sided rowhammer attack, *CLFLUSH_hammer*, adapted from [106]. The second application is CLFLUSH-free double-sided rowhammer attack, *CLFLUSH-free_hammer*, used to demonstrate our CLFLUSH-free attack in Section 2. We also measure the slowdown incurred on non-malicious programs using SPEC2006 integer benchmarks [107] and analyze our detection algorithm's sensitivity to potential future attacks.

4.4.2 Rowhammer Detection Characteristics

We first evaluate ANVIL's ability to detect rowhammer activity. The evaluation is done for scenarios where the test machine is heavily and lightly loaded. To emulate heavy load, we run the rowhammering applications along with memory-intensive applications (*mcf*,

LLC_MISS_THRESHOLD	20K
Miss Count Duration (t_c)	6ms
Sampling Duration (t_s)	6ms

Table 4.2 Rowhammer Detector Parameters to Evaluate Accuracy of Rowhammer Detection

libquantum and *omnetpp* running at the same time) from the SPEC2006 integer benchmark suite. The detector parameters used for our evaluation are given on Table 4.2.

The time values are selected to be low enough so that any rowhammering activity can be detected with enough time to deploy protection. With this setting, hammering activity can be detected within 12 milliseconds. The last-level cache miss threshold value was experimentally found by considering the minimum number of memory accesses required to cause a DRAM bit flip. In our experiments the minimum number of memory accesses that caused a flip was 220K for CLFLUSH-based double-sided rowhammering attack. In order to achieve this many activations within a refresh period of 64ms, a minimum of 20.6K activations must occur within 6ms. Therefore, we will use 20K misses in 6ms as a threshold value for the first stage of detection.

Table 4.3 shows the result of rowhammering detection for applications *CLFLUSH_hammer* and *CLFLUSH-free_hammer* under heavy and light load. For both attacks, the table shows the average time to detect a rowhammer attack within a 64ms refresh cycle in which rowhammering was occurring. The table also lists the average selective refresh rate, which are refreshes that occur when the rowhammer detector identifies potential DRAM victim rows. As seen in these results, ANVIL is quite responsive, with response times well within a single refresh cycle, and with only slight increases in response time due to a heavy loaded system. In addition the selective refresh rates are low, but sufficient for multiple refreshes within a single refresh cycle for any detected victim row. The low selective refresh rate ensures that a clever attacker cannot use the selective refresh to hammer other DRAM rows. Finally, it is good to note that our detector stopped all rowhammering, resulting in zero bit flips for all of the attacks.

Benchmark	Average Time to Detect	Refreshes per 64ms	Total Bit Flips
CLFLUSH *(Heavy Load)*	12.8 ms	12.35	0
CLFLUSH *(Light Load)*	12.3 ms	10.3	0
CLFLUSH-free *(Heavy Load)*	35.3 ms	4.53	0
CLFLUSH-free *(Light Loaded)*	22.85 ms	5.10	0

Table 4.3 Rowhammer Detection Result for Rowhammering Programs: The table shows the average time before a rowhammer activity is detected and the rate of selective refreshes performed.

4.4.3 Performance Evaluation

We evaluated the slowdown incurred by ANVIL by analyzing the execution of non-malicious applications from the SPEC2006 integer benchmark suite. We used the parameters listed on Table 4.2 for the evaluation. In addition to this experiment, we compare the performance overhead of ANVIL with that incurred by doubling DRAM refresh rate. For these evaluations our baseline is an unprotected system with a refresh period of 64ms.

Figure 4.3 shows relative execution times for ANVIL-protected system relative to our baseline. The ANVIL-protected system has peak and average overheads of 3.18% and 1.17%, respectively. Most of the performance overhead by ANVIL is attributed to the low last-level cache miss rate threshold. *libquantum*, *omnetpp*, *mcf* and *Xalancbmk* crossed the last-level cache miss threshold 95% to 99% of the time. On the other extreme, *h264ref*, *gobmk*, *sjeng* and *hmmer* crossed the threshold less than 10% of the time. This indicates that sampling of addresses in the second stage of the detection phase contributes to almost all of the performance overhead. Clearly, the overheads of continuously running ANVIL's rowhammer detection are very low. Low enough to protect existing systems from rowhammer attacks, and likely low enough to obviate the need for dedicated hardware-based rowhammer protection mechanisms in future systems.

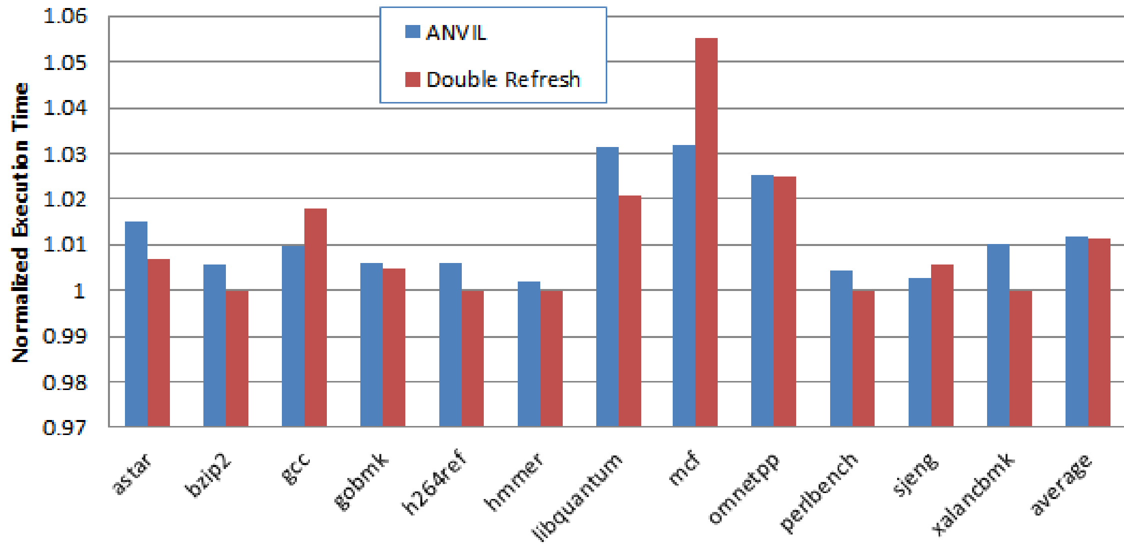


Figure 4.3 ANVIL's Impact on Non-Malicious Programs: The Figure shows execution times for selected benchmarks on ANVIL-enabled system and a system with doubled DRAM refresh rate. The values shown are normalized to execution time without ANVIL and at a single refresh period.

Table 4.5 shows the false positive rate for the SPEC2006 integer benchmarks. The rate is measured as the average number of superfluous selective refreshes per second. The number of false positives is low enough that selective refresh of rows has negligible effect on performance.

4.4.4 Comparison with Increased Refresh Rate

As we have shown in Section 2, doubling DRAM refresh rate is not sufficient to prevent all rowhammering attacks. Equally important is the execution time and power overhead incurred by the increased refresh rate. Previous studies have shown that increasing refresh rate reduces parallelism in the memory subsystem, affecting overall system performance [108, 109]. Figure 4.3 shows performance overhead of doubling DRAM refresh rate as compared with our software protection mechanism. ANVIL's performance overheads are only marginally larger (on average) than doubling the refresh rate, while providing a significantly higher level of protection against rowhammer attacks, as demonstrated in Section 2. As can

Benchmark	Refreshes/sec
astar	0.10
bzip2	1.05
gcc	0.71
gobmk	0.19
h264ref	0.00
hmmer	0.00
libquantum	0.06
mcf	0.01
omnetpp	0.02
perlbench	0.00
sjeng	0.00
Xalancbmk	0.05

Table 4.4 Rate of False Positive Refreshes: The table shows rate of superfluous refreshes for SPEC2006 integer benchmarks while running under ANVIL.

be observed, memory intensive applications like *mcf* suffer most from doubling DRAM refresh rate thus, their performance benefits greatly from the use of ANVIL's protection.

4.4.5 Robustness to Potential Future Rowhammer Attacks

As the density of DRAM devices increase, DRAM cells become more susceptible to disturbance errors. It is then expected that for future DRAM devices, rowhammer attacks will be possible with less DRAM row activations. An attacker might take advantage of this to evade detection by our software protection mechanism by: 1) Activating DRAM aggressor rows at a high rate such that rowhammer attacks will be faster than they can be detected by the protection mechanism. 2) Spreading out fewer DRAM row activations over a refresh period such that the last-level cache miss rate stays below the last-level cache miss threshold. Our detection mechanism can cope with both situations by adjusting the detector parameters listed on Table 4.2. To evaluate the effect that more nimble future attacks have on the performance of non-malicious programs, we consider a future scenario where bit flips

can occur with 110K DRAM row accesses (i.e., half the number of accesses that produced flips on our experiments).

Figure 4.4 examines the performance impact on a subset of the SPEC2006 benchmarks for three cases. The benchmarks are selected to be representatives of the memory access characteristics of SPEC2006 benchmark suit. *ANVIL-baseline* is our baseline detector with parameters as given on Table 4.2. *ANVIL-heavy* considers the case where the 110K DRAM row accesses can occur within 7.5ms (i.e., half the time we observed for our experiments). For this case values of t_c and t_s are set to 2ms while the value of the last-level cache miss threshold remains unchanged at 20K. The third case, *ANVIL-light*, considers a situation where the 110K DRAM row accesses are spread out across a refresh period of 64ms (i.e., half the number of accesses purposely spread out maximally). For this case values of t_s and t_c are set to 6ms, and the last-level cache miss threshold is halved to 10K. As seen in Figure 4.4, ANVIL has room to grow if future rowhammer attacks become more aggressive. Overheads do grow to detect these more nimble attacks, but only slightly. Increasing the last-level miss sample period to 2ms has the larger performance impact, which is expected as this is the first-stage mechanism, whose performance overheads are experienced continuously.

Table 4.5 shows false positive refresh rates due to false positives for *ANVIL-light* and *ANVIL-heavy*. Though both configurations show an increase in false positive rates than *ANVIL-baseline*, they do not incur significant overheads.

4.5 Related Work

Previous works have studied exploitation and prevention of the rowhammer vulnerability. In this section, we detail currently known attacks and possible mitigations.

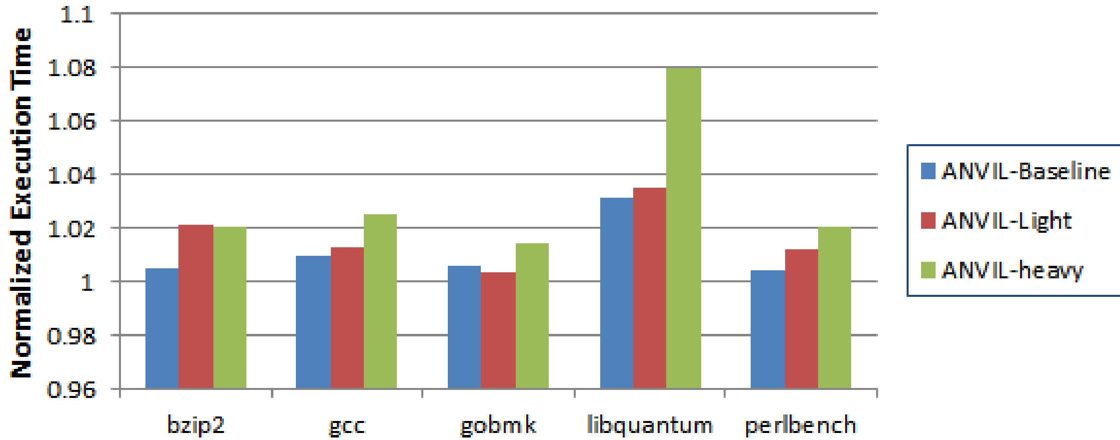


Figure 4.4 Sensitivity of Execution Overheads to Potential Future Attacks: The Figure compares normalized execution times for selected benchmarks running on *ANVIL-enabled* system with three configurations. *ANVIL-Heavy* is configured to have the highest sampling rate while *ANVIL-Light* has the lowest last-level cache miss threshold.

Benchmark	Refreshes/sec (ANVIL-light)	Refreshes/sec (ANVIL-heavy)
bzip2	1.61	1.09
gcc	7.12	1.88
gobmk	0.28	0.84
libquantum	0.13	0.08
perlbench	0.06	0.00

Table 4.5 Rate of False Positive Refreshes for *ANVIL-Heavy* and *ANVIL-Light*: The table shows false positive rates for selected SPEC2006 integer benchmarks while running under two ANVIL configurations. *ANVIL-Heavy* has a relatively small sampling period which reduces the probability of misses with high address locality on non-malicious applications. On the other hand, *ANVIL-light* allows more samples for a longer sampling period thus resulting in a relatively larger false positive rate.

4.5.1 Rowhammer Vulnerability and Its Exploitation

Even if the rowhammer vulnerability on modern DRAMs has been known by manufacturers since at least 2012 [36], the first detailed experimental study was published in 2014 by Yoongu Kim, et. al [36]. Their study shows that bits in a DRAM row (the victim row) can be flipped by repeatedly accessing adjacent rows in the same bank (the aggressor rows). The authors used x86's CLFLUSH instruction to bypass the cache and enable frequent references

directly to DRAM.

Leveraging the early attack demonstrations, Seaborn and Dullien [37] demonstrated two security exploits that take advantage of rowhammer-induced bit flips. Their first attack bypasses Google's Native Client (NaCl) sandboxing system. NaCl is a software sandbox, integrated with the Chrome browser, that allows secure execution of untrusted client side applications and plug-ins. NaCl works by carefully scanning code for illegal code operations at load time (e.g., system calls or arbitrary indirect jumps), and by funneling all I/O operations through a security analyzer. The NaCl rowhammer attack works by having a securely loaded NaCl application hammer its own code segment until an illegal arbitrary code jump sequence is formed, then the application jumps to the middle of an instruction where illegal operations can be formed from validated code. Note that the attack is changing code that has been verified and deemed safe. Since the instructions are modified at the hardware level, the sandbox will not be aware of any of these changes. The author's current proof-of-concept implementation can take advantage of 13% of the possible bit flips within an instruction.

Seaborn and Dullien's second attack takes advantage of the bit flips to bypass the memory page protection mechanism of a Linux system running on x86-64 [37]. The attack works by filling physical memory with page tables for a single process, by repeatedly mmap()'ing a file into its memory. This repeated file mapping sprays the memory with PTEs that are used to translate the newly mmap()'ed virtual addresses. By rowhammering the memory with page tables, there is a non-trivial probability that a PTE will be changed to point to a physical page containing a page table, thereby giving the application access to its own page tables. This will give the attacker full R/W permission to a page table entry, which in effect results in access to all of physical memory.

Even if all of the exploits mentioned above rely on CLFLUSH instruction in x86, the attack we presented in Section 2 demonstrated how rowhammer attacks can be launched without the use of any cache line flush instruction. Another CLFLUSH-free rowhammer attack announced in July 2015 showed that it is possible to cause flips from within a JavaScript

application running in a browser [42]. It is important to note that our CLFLUSH-free attack was completed in May 2015.

4.5.2 Rowhammer Mitigations

In this section, we detail both software and hardware techniques that have been proposed and deployed to protect against data corruption and security exploits due to rowhammering.

Protections for Legacy Systems

Software Patches: To date, two open source projects have released patches in response to the security vulnerabilities explained above. Google's NaCl sandbox was patched to disallow the use of CLFLUSH instruction by applications running inside it. The attack mechanism we presented in Section 2 defeats this protection and enables malicious applications to effectively hammer rows without using CLFLUSH (or any other explicit cache flush instructions).

Recently, the Linux kernel was updated to disallow the use of the pagemap interface from the user space, as a measure to make it more difficult to do double-sided rowhammering in Linux-based systems. This change prevents malicious applications from analyzing the physical address space to launch targeted attacks. However, this attack still leaves room for potential attacks that rely on side-channel information to make inferences about the physical memory layout. Furthermore, certain attacks such as the NaCl sandbox escape attack can be implemented by repeatedly picking two random addresses without having any knowledge of the physical address mapping.

Doubling Refresh Rate: Some vendors published BIOS updates that double DRAM refresh rates(i.e. halving the refresh interval from 64ms to 32ms) [38, 39, 40]. Doubling the refresh rate reduces the amount of time an attacker has to mount an attack, since the discharging of a rowhammer'ed bit must be completed within one refresh cycle. However,

our empirical studies show that it is still possible to induce bit flips through double-sided hammering even when the refresh period is as low as 16ms. Further increases in refresh rates would have significant effects on system performance and energy consumption [36].

Protections for Future Systems

Currently available hardware-based reliability features are not capable of mitigating DRAM disturbance errors. Error Correcting Codes(ECC) protection, aside from being expensive, is capable of repairing single-bit flips only. Furthermore, ECC will turn the problem of bit-flips into denial of service if the system has to deal with machine check exceptions every time a flip is detected [37].

Due to the inability of current memory controllers and memory modules to deal with rowhammer attacks, multiple hardware enhancements have been proposed. The possibility of having an activation counter for each row in a DRAM module has been considered in literature [46, 36]. However, due to the high overhead of maintaining and updating per-row counters, other alternatives have been recommended.

Probabilistic row refreshing has been proposed as an alternative to per-row counters [46, 36]. In this technique, when activation command is sent to a row, a random number generator is used to decide if adjacent row has to be refreshed. Since requests to rows that are being hammered will be encountered very frequently, there is a high probability that it will trigger a refresh.

Project Armor [33] introduces an extra buffer that will cache data from rows with repeated activation commands. By servicing requests to hammered rows from the extra buffer, Armor DRAM prevents rows from being accessed repeatedly.

Processor and memory manufacturers are also deploying products with capabilities to perform targeted row refreshes. The current LPDDR4 standard and recent DDR4 modules support targeted refresh of potential victim rows [43, 44]. Intel has published patents on memory controllers that support targeted row refresh [110]. The memory controllers are

designed to identify repeated reads to a row. However, the actual physical placement of rows can differ among different manufacturers. Hence, the controller only transmits the row that is being repeatedly accessed, and the memory module is responsible for refreshing the victim rows based on its internal structure.

It is important to note that the mitigation techniques for existing systems (i.e., doubling refresh rate and removing access to CLFLUSH) are shown to be ineffective in this work. We are able to implement the rowhammer attack in a 32ms double-rate refresh cycle, and we can also rowhammer DRAMs without access to the CLFLUSH instruction. While newly proposed hardware enhancements can protect future systems from rowhammer attacks, a software solution is still necessary to protect current hardware. As such, in this chapter we detail a low-cost software-based rowhammer detector that thwarts attacks with little performance impact. It is our claim that these protections are appropriate both for existing and future designs.

4.6 Chapter Summary

In this chapter we systematically analyzed a security vulnerability found in commodity DRAM chips referred to as rowhammer. Rowhammer attacks use the CLFLUSH instruction to accomplish hammering by bypassing processor caches and repeatedly accessing memory.

We demonstrated that existing mitigation techniques such as doubling refresh rates and disallowing CLFLUSH instructions are not sufficient—we showed that it possible to rowhammer in as little as 15ms. We also showed the first CLFLUSH-free rowhammer attack that does not require special cache flushing instructions, therefore expands the rowhammering attack surface. As an alternative protection mechanism, we designed, implemented and evaluated ANVIL, the first software-based defense that protects against rowhammer attacks. Our defense leverages the insight that rowhammer memory access patterns are fundamentally different from those of normal applications. Compared to prior approaches,

ANVIL is more effective, has lower cost, is readily deployable and is adaptable due to its software-based approach. Experiments with a diverse set of benchmarks on a real system showed that ANVIL has an average slowdown of 1% and less than 1% false positive detections, while protecting against all tested rowhammer attacks.

Chapter 5
Conclusion

In this dissertation, we showed how modern hardware features can be leveraged to provide novel and efficient system security solutions. In the first part of the dissertation we showed how we can improve the performance of memory safety techniques using existing hardware features. Chapter 2 presented an efficient temporal memory safety technique that takes advantage of pointer authentication feature. Our proposed technique uses the unused bits of a pointer to store metadata. This reduces the memory required to store metadata while allowing faster metadata access. Our technique reduces the memory overhead by 90% as compared to a software-only solution. In Chapter 3, we presented an efficient sandboxing mechanism for low-end embedded systems. This mechanism uses a widely-available memory protection unit hardware along with a small runtime to provide efficient sandboxing mechanism. It doesn't require any hardware changes and incurs a performance overhead of a little more than 1%.

In the second part of the dissertation, we presented a novel technique to detect rowhammer attacks using existing hardware performance counter features. Our technique doesn't require any hardware changes and has an average slowdown of only 1%.

5.1 Future Directions

We expect processors to continue adding new features, presenting opportunities to provide efficient solutions to problems not addressed in this dissertation. However, there are many opportunities to expand on the solutions presented on this dissertation. We list a few below.

Complete memory safety: Currently, PETS (presented in Chapter 2) only detects temporal errors such as use-after-free errors. As discussed in Chapter 1, to provide complete memory safety, we need to address the problem of spatial memory safety as well. An interesting future work would be extending PETS to provide spatial memory safety as well. In particular, Low-fat PETS, a version of PETS that uses a low-fat memory allocator, can easily be extended to support spatial safety without requiring any additional metadata storage. Spatial safety techniques typically involve checking the bounds of memory accesses. Since the low-fat memory layout allows implicitly storing the size and base address information of a buffer, bounds checking can be performed from a pointer value alone [49]. Bounds checking can be added on top of the current checks done by PETS.

Leveraging CPU features in other temporal safety techniques: PETS is based on the lock-and-key temporal safety technique. However, as discussed in Chapter 2, there are other approaches of providing temporal safety. One approach is the page-permission-based approach [10, 59]. Oscar [10] assigns new virtual pages for each memory allocation. On free, the corresponding virtual pages are made inaccessible by changing their permissions. Dangling pointer accesses are then detected through page faults. Even though this approach has a lower performance overhead as compared to the lock-and-key technique, applications that allocate/deallocate memory frequently can experience a high performance overhead. Part of the overhead comes from the system calls when creating and disabling shadow pages [10]. These expensive system calls can be avoided by using hardware features such as Intel's Protection Keys [103]. These features allow assigning page-permissions to virtual pages in *user mode*, obviating the need for system calls.

Addressing other Vulnerabilities/Challenges: Existing hardware features can also be adopted to address other security challenges not discussed in this dissertation. They can be used to either improve performance of existing solutions or provide new approaches. One area of interest is side-channels. Some works have already proposed new ways of

using existing processor features to defend against timing side-channel attacks [111, 112]. However, there are other types of side-channels not addressed by these solutions, such as speculative execution [113, 114]. It would be interesting to explore ways of providing protections against such types of side-channels using existing processor features.

Program anomaly detection is another security technique that can benefit from existing processor features. Program anomaly detection is a general technique that identifies abnormal program behaviours (*i.e.* behaviours that do not fit into normal program behaviors) caused by attacks. Anomaly detection typically involves collecting and analyzing program traces. Collecting program traces can be accelerated by using hardware features such as Intel PT [115]. In addition to this, there is a growing interest in using microarchitectural characteristics of programs for anomaly detection [116, 117]. ANVIL, presented in Chapter 4 of this dissertation, is also an example of anomaly detection using microarchitectural characteristics. Modern hardware performance counters are equipped with features that provide detailed microarchitectural characteristics of programs. Leveraging these features to detect other attacks would be an interesting future work.

Bibliography

[1] The Verge. *UK hospitals hit with massive ransomware attack*. May 2017.

[2] zdnet. *A critical security flaw in popular industrial software put power plants at risk*. May 2018.

[3] zdnet. *WhatsApp fixes bug that let hackers take over app when answering a video call*. October 2018.

[4] wccftech. *Privilege Elevation, Arbitrary Code Execution, DoS Apples Latest iOS 12.1 Brings Fixes to Several Critical Security Issues*. October 2018.

[5] Cve details. `https://www.cvedetails.com/`.

[6] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM.

[7] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986, May 2016.

[8] 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, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society.

[9] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Cets: Compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 31–40, New York, NY, USA, 2010. ACM.

[10] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 815–832, Vancouver, BC, 2017. USENIX Association.

[11] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 405–419, New York, NY, USA, 2017. ACM.

[12] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[13] Yves Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[14] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 290–301, New York, NY, USA, 1994. ACM.

[15] 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)*, pages 147–163, Broomfield, CO, 2014. USENIX Association.

[16] PaX Team. PaX address space layout randomization (ASLR). `http://pax.grsecurity.net/docs/aslr.txt`, 2003.

[17] Santosh G. Nagarakatte. Practical low-overhead enforcement of memory safety for c programs. 2012.

[18] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 189–200, Washington, DC, USA, 2012. IEEE Computer Society.

[19] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 133:1–133:6, New York, NY, USA, 2014. ACM.

[20] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. Hcfi: Hardware-enforced control-flow integrity. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 38–49, New York, NY, USA, 2016. ACM.

[21] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[22] Intel Inc. *Control-flow Enforcement Technology Preview*. June 2017.

[23] Andrew Baumann. Hardware is the new software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 132–137, New York, NY, USA, 2017. ACM.

[24] The results - pown2own 2017 day three. `https://www.thezdi.com/blog/2017/3/17/the-results-pwn2own-2017-day-three`.

[25] Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of c programs. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 117–126, New York, NY, USA, 2004. ACM.

[26] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. SoK: Sanitizing for Security. *ArXiv e-prints*, June 2018.

[27] Inc. Qualcomm Technologies. Pointer authentication on armv8.3., January 2017.

[28] Anthony J. Bonkoski, Russ Bielawski, and J. Alex Halderman. Illuminating the security issues surrounding lights-out server management. In *Proceedings of the 7th USENIX Conference on Offensive Technologies*, WOOT'13, pages 10–10, Berkeley, CA, USA, 2013. USENIX Association.

[29] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 95–110, San Diego, CA, 2014. USENIX Association.

[30] Project Zero. Over The Air: Exploiting Broadcom's Wi-Fi Stack. `https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html`.

[31] ARM Limited. *ARM Cortex-R Series Programmer's Guide*. 2014.

[32] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 10:1–10:14, New York, NY, USA, 2014. ACM.

[33] Jim Garside Mohsen Ghasempour, Mikel Lujan. Armor: A Run-Time Memory Hot-Row Detector. `http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer/index.html`.

[34] Ram Kumar, Eddie Kohler, and Mani Srivastava. Harbor: Software-based memory protection for sensor nodes. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, IPSN '07, pages 340–349, New York, NY, USA, 2007. ACM.

[35] R. Kumar, A. Singhania, A. Castner, E. Kohler, and M. Srivastava. A system for coarse grained memory protection in tiny embedded processors. In *2007 44th ACM/IEEE Design Automation Conference*, pages 218–223, June 2007.

[36] Yoongu Kim, R. Daly, J. Kim, C. Fallin, Ji Hye Lee, Donghyuk Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 361–372, June 2014.

[37] Thomas Dullien Mark Seaborn. Exploiting the DRAM rowhammer bug to gain kernel privileges. March 2015.

[38] Apple Inc. About the security content of mac EFI security update 2015-001. https://support.apple.com/en-us/HT204934.

[39] HP Inc. HP Moonshot Component Pack Version 2015.05.0. http://h17007.www1.hp.com/us/en/enterprise/servers/products/moonshot/component-pack/index.aspx.

[40] Lenovo Inc. Row Hammer Privilege Escalation Lenovo Security Advisory: LEN-2015-009. https://support.lenovo.com/us/en/product_security/row_hammer.

[41] M. T. Aga, Z. B. Aweke, and T. Austin. When good protections go bad: Exploiting anti-dos measures to accelerate rowhammer attacks. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 8–13, May 2017.

[42] D. Gruss, C. Maurice, and S. Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. *ArXiv e-prints*, July 2015.

[43] JEDEC Solid State Technology Association. Low power double data rate 4 (LPDDR4), August 2015.

[44] Micron Inc. *DDR4 SDRAM MT40A2G4, MT40A1G8, MT40A512M16 Data sheet*. 2015.

[45] Mark Lanteigne. How rowhammer could be used to exploit weaknesses in computer hardware, March 2016.

[46] Dongkyu Kim, Prashant Nair, and Moin Qureshi. Architectural support for mitigating row hammering in DRAM memories. IEEE, 2015.

[47] Roberto Avanzi. The QARMA block cipher family - almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes. *IACR Cryptology ePrint Archive*, 2016:444, 2016.

[48] ARM Limited. *ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile*. 2017.

[49] Gregory J. Duck and Roland H. C. Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 132–142, New York, NY, USA, 2016. ACM.

[50] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[51] Lowfat: Lean c/c++ bounds checking with low-fat pointers. `https://github.com/GJDuck/LowFat`.

[52] Qemu emulator. `https://www.qemu.org/`.

[53] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[54] Xilinx zynq ultrascale+ mpsoc zcu102 evaluation kit. `https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html`.

[55] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 133–143, New York, NY, USA, 2012. ACM.

[56] Intel. Pointer checker overview. `https://software.intel.com/en-us/node/522703`.

[57] Bruce Perens. *Electric fence malloc debugger*. 1993.

[58] How to use pageheap.exe in windows xp, windows 2000, and windows server 2003.

[59] Dinakar Dhurjati and Vikram Adve. Efficiently detecting all dangling pointer uses in production servers. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pages 269–280, Washington, DC, USA, 2006. IEEE Computer Society.

[60] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.

[61] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 213–223, Washington, DC, USA, 2011. IEEE Computer Society.

[62] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[63] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM.

[64] Gene Novark and Emery D. Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 573–584, 2010.

[65] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, Boston, MA, 2012. USENIX.

[66] C. Miller and C. Valasek. *Remote Exploitation of an Unaltered Passenger Vehicle*. 2015.

[67] B. Krebs. DDoS on Dyn Impacts Twitter, Spotify, Reddit. `https://krebsonsecurity.com/2016/10/`.

[68] Aurélien Francillon and Claude Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 15–26, New York, NY, USA, 2008. ACM.

[69] ARM Limited. *ARMv7-M Architecture Reference Manual*. December 2014.

[70] radare2. `https://github.com/radare/radare2`.

[71] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 3–14, Dec 2001.

[72] STMicroelectronics. NUCLEO-F446RE. `http://www.st.com/en/evaluation-tools/nucleo-f446re.html/`.

[73] NXP. *FRDM-K64F:Freedom Development Platform for Kinetis K64, K63, and K24 MCUs*.

[74] GitHub. mbed TLS. `https://github.com/ARMmbed/mbedtls`.

[75] Freertos. `http://www.freertos.org/`.

[76] ARM Limited. *ARM Cortex-M4 Processor Technical Reference Manual*. 2015.

[77] ARM Limited. Wearable Reference Design. `https://www.mbed.com/en/technologies/applications/wearables/`.

[78] GitHub. mbed TLS. `https://github.com/ARMmbed/mbed-tls-sockets/tree/master/test/tls-client`.

[79] Myotest. `http://www.myotest.com/`.

[80] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[81] Bennet Yee, David Sehr, Greg Dardyk, Brad Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy (Oakland'09)*, IEEE, 3 Park Avenue, 17th Floor, New York, NY 10016, 2009.

[82] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary cpu architectures. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.

[83] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association.

[84] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 558–569, New York, NY, USA, 2014. ACM.

[85] F. Brasser, B. El Mahjoub, A. R. Sadeghi, C. Wachsmann, and P. Koeberl. Tytan: Tiny trust anchor for tiny devices. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2015.

[86] ARM Limited. mbed uVisor. `https://www.mbed.com/en/technologies/security/uvisor/`.

[87] ARM Limited. *TrustZone Technology for ARM v8-M Architecture*. 2016.

[88] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer. Protecting bare-metal embedded systems with privilege overlays. In *Proceedings of the 38th IEEE Symposium on Security and Privacy*. IEEE, 2017.

[89] Daniele Midi, Mathias Payer, and Elisa Bertino. Memory safety for embedded devices with nescheck. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 127–139, New York, NY, USA, 2017. ACM.

[90] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-flat: Control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 743–754, New York, NY, USA, 2016. ACM.

[91] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 479–498, Washington, D.C., 2013. USENIX.

[92] T.C. May and Murray H. Woods. Alpha-particle-induced soft errors in dynamic memories. *Electron Devices, IEEE Transactions on*, 26(1):2–9, Jan 1979.

[93] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: A large-scale field study. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 193–204, New York, NY, USA, 2009. ACM.

[94] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 297–310, New York, NY, USA, 2015. ACM.

[95] Vilas Sridharan and Dean Liberty. A study of dram failures in the field. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 76:1–76:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[96] 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*, SP '03, pages 154–, Washington, DC, USA, 2003. IEEE Computer Society.

[97] J. Xu, S. Chen, Z. Kalbarczyk, and R.K. Iyer. An experimental study of security vulnerabilities caused by errors. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pages 421–430, July 2001.

[98] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 191–205, May 2013.

[99] L3 Cache Mapping on Sandy Bridge CPUs.
http://lackingrhoticity.blogspot.com/2015/04/l3-cache-mapping-on-sandy-bridge-cpus.html.

[100] Intel Inc. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. September 2014.

[101] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 60–71, New York, NY, USA, 2010. ACM.

[102] Paul J. Drongowski. Instruction-based sampling: A new performance analysis technique for AMD family 10h processors. In *Advanced Micro Devices Inc*, 2007.

[103] Intel Inc. *Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3 (3A, 3B & 3C): System Programming Guide*. June 2015.

[104] 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 '16, pages 1675–1689, New York, NY, USA, 2016. ACM.

[105] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of rowhammer defenses. *CoRR*, abs/1710.00551, 2017.

[106] Program for Testing for the DRAM "rowhammer" Problem.
https://github.com/mseaborn/rowhammer-test.

[107] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[108] Ishwar Bhati, Mu-Tien Chang, Zeshan Chishti, Shih-Lien Lu, and Bruce Jacob. Dram refresh mechanisms, penalties, and trade-offs. In *IEEE TRANSACTIONS ON COMPUTERS, VOL. 64*, 2015.

[109] Janani Mukundan, Hillery Hunter, Kyu-hyoun Kim, Jeffrey Stuecheli, and José F. Martínez. Understanding and mitigating refresh overheads in high-density ddr4 dram systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 48–59, New York, NY, USA, 2013. ACM.

[110] K. Bains, J.B. Halbert, C.P. Mozak, T.Z. Schoenborn, and Z. Greenfield. Row hammer refresh command, January 2014. WO Patent App. PCT/US2013/048,016.

[111] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–418, March 2016.

[112] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 217–233, Vancouver, BC, 2017. USENIX Association.

[113] Paul Kocher, Jann Horn, Anders Fogh, , 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 *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[114] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[115] Intel. Processor tracing. `https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing`.

[116] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore J. Stolfo. On the feasibility of online malware detection with performance counters. In *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, 2013.

[117] M. Kazdagli, V. J. Reddi, and M. Tiwari. Quantifying and improving the efficiency of hardware-based mobile malware detectors. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.