Defeating Code Reuse Attacks with Minimal Tagged Architecture

by

Samuel Fingeret

B.S., Massachusetts Institute of Technology (2014)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2015

Author .....
Department of Electrical Engineering and Computer Science
September 16, 2015

Certified by .....
Prof. Howard Shrobe
Principal Research Scientist MIT CSAIL.
Thesis Supervisor

Accepted by .....
Prof. Christopher Terman
Chairman, Masters of Engineering Thesis Committee

Defeating Code Reuse Attacks with Minimal Tagged Architecture

by

Samuel Fingeret

Submitted to the Department of Electrical Engineering and Computer Science
on September 16, 2015, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

In this thesis, we argue that software-based defenses to code reuse attacks are fundamentally flawed. With code pointer integrity as a case study, we show that a secure and efficient software-based defense to code reuse attacks is impossible and thus motivate the case for hardware approaches. We then propose our tagged architecture system Taxi (Tagged C) as a practical defense against code reuse attacks which minimally modifies existing hardware components. We also propose strong defense policies which aim to guarantee security while minimizing tag memory usage. Our Taxi prototype, a modified RISC-V ISA simulator, demonstrates that we can defeat code reuse attacks with high compatibility and low memory overhead.

Thesis Supervisor: Prof. Howard Shrobe
Title: Principal Research Scientist MIT CSAIL.

Acknowledgments

My sincerest thanks to my advisor Howard Shrobe for his insight and guidance as well as my lab partner Julián González for his knowledge and for keeping me on track throughout the past year. I am also grateful to Isaac Evans, Ulziibayar Otgonbaatar, Tiffany Tang and especially Stelios Sidiroglou-Douskos and Hamed Okhravi for their research discussions and feedback. Without them, this work would not be possible.

This work is sponsored by the Office of Naval Research under the award N00014-14-1-0006, entitled Defeating Code Reuse Attacks Using Minimal Hardware Modifications. Opinions, interpretations, conclusions and recommendations are those of the author and do not reflect official policy or the position of the Office of Naval Research or the United States Government.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Although buffer overflow vulnerabilities have been exploited for over twenty years [34] and can lead to remote code execution, modern mitigation techniques remain ineffective due to the rise of code reuse attacks such as return oriented programming [41]. These attacks chain together previously existing code in the form of gadgets which can be combined to execute a malicious payload, allowing attackers to bypass defenses such as write xor execute which prevent injection of new code [49].

The problem of defense then reduces to distinguishing intended code behavior from malicious code behavior, which is a much more difficult problem than distinguishing code loaded in at program startup from code loaded during program execution. We must also do this efficiently: C is primarily used for its low-level performance benefits, so defenses with high overhead are impractical.

Unfortunately, existing software-based defenses are bypassable with the exception of full memory safety [31], though memory safety requires prohibitively high performance and memory overhead. Defense techniques have tried enforcing restrictions on gadgets [33] [36] [19], but the gadget space is diverse [5] [37] and malicious payloads still exist even with these constraints [6] [38] [14]. Code diversification defenses randomize code locations in the executable so that attackers cannot locate desired gadgets [46] [24] [52] [35] [23], but disclosure vulnerabilities and side-channel attacks render code diversification ineffective [4] [40]. Finally, defenses which enforce static constraints on control flow [2] [56] [55] [30] [12] [47] are believed to be too permissive

in order to maintain compatibility with existing code, so exploits are still possible [21].

A promising defense known as Code Pointer Integrity or CPI attempts to enforce memory safety on only code pointers [27]. Because code pointers account for a small fraction of all pointers in an application, CPI's overhead is significantly less than that of full memory safety. However, implementing CPI is difficult because data pointer corruption is allowed under this defense, so an attacker could corrupt any addressin memory, including CPI's internal state. On 64-bit systems, CPI protects its internal state using information hiding through randomization, which we show to be ineffective [17]. The failure of CPI to protect its internal state suggests a fundamental flaw in software-based defenses in that efficient protection of defense metadata is difficult.

We thus turn to hardware-based defenses to improve efficiency and security of code reuse defenses. In particular, we look at tagged architectures, or systems which attach a small amount of metadata, called a tag, to every word of memory. Tagged architectures are believed to be both secure and efficient [15] [16] [9], but memory overhead and lack of compatibility prevent tagged architectures from being adopted. We design our own tagged architecture system called Taxi (short for tagged C) with the goal of minimizing the number of tag bits and minimizing modifications to existing hardware components. We choose the open-source RISC-V architecture as our base platform [54].

To prevent code reuse attacks, Taxi adds a tag bit to every word of memory and this bit is used to mark code pointers. Taxi initializes and propagates tags transparently, so even an attacker with control over an application's memory cannot forge tags and divert control flow through code pointer corruption. Although Taxi is still in the ISA simulator stage, our prototype is capable of executing real-world programs inside a cross-compiled Linux kernel and this allows us to verify security guarantees as well as compatibility with existing code.

We now describe the contributions of this thesis. In chapter 2 we present a brief analysis of previous software-based defenses as well as known methods for bypassing them. In chapter 3 we demonstrate an attack against CPI's use of randomization

to hide code pointers and demonstrate that randomization is insufficient to protect internal defense state. In chapter 4 we introduce our tagged architecture model and compare our ideal implementation to previous tagged architecture defenses. In chapter 5 we describe the design and implementation of Taxi in our RISC-V ISA simulator and in chapter 6 we evaluate Taxi on real-world programs with the conclusion that a hardware-based implementation of Taxi is practical and secure. In chapter 6 we also provide insight into problematic code patterns which make tagged architecture implementations more difficult. In chapter 7 we conclude and discuss future improvements to Taxi.

Chapter 2

Background

2.1 Code Reuse Attacks

2.1.1 Buffer Overflows

A buffer overflow, or spatial memory violation, occurs when a program attempts to write data past the bounds of a buffer due to a programming bug, causing adjacent data to be overwritten. Frequently, this buffer lies on the stack, so the overwritten data consists of local variables, return addresses, and register values from previous stack frames, thus changing the behavior of the program in unintended ways. This usually leads to a crash, but when exploited by an attacker, the input is crafted such that a return address or local function pointer is overwritten by a value chosen by the attacker, allowing the attacker to control which code gets executed [34].

Return addresses, function pointers, and other pointers which point to a code address are the main targets of an overflow because they allow the attacker to directly redirect control flow, and we refer to these as code pointers. Even if such a vulnerability is on the heap rather than on the stack, the attacker can still modify nearby variables and maintenance pointers used by the memory allocator, and this is also enough to let the attacker write malicious data anywhere in memory [13].

Modern compilers attempt to mitigate stack overflows by adding random canary values to the stack between local variables and return addresses. These stack canaries

have their value checked on return, so if an attacker tries to overflow into a return address, the canary will be overwritten as well and the overflow will be detected [10]. This defense can be bypassed with a memory disclosure vulnerability that leaks the canary value or by overflowing into a function pointer local variable rather than a return address. The defense can also be bypassed by overflowing an exception handling structure on the stack, since an incorrect canary value causes an exception which will then be handled by code at a location of the attacker's choice [44].

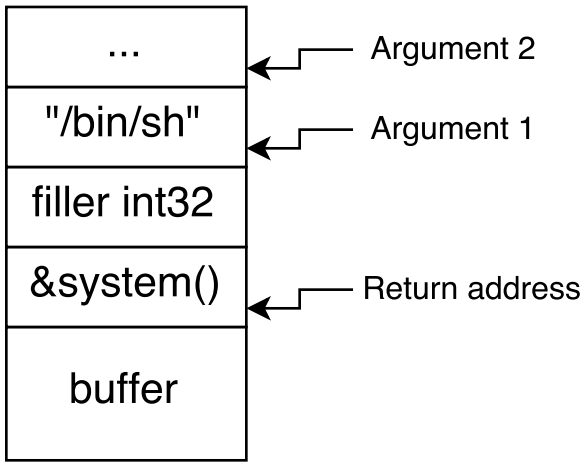
2.1.2 Use After Free

A use after free occurs when a program attempts to use memory after it has been freed, also due to a programming bug. If an attacker can modify the data using the freed pointer, then once the freed memory is reallocated, the attacker will be able to overwrite the new structure and again cause unexpected behavior. A recent exploit using this technique allocates a variable size array using the freed memory and uses the dangling pointer to corrupt the length field, thus creating a buffer overflow and allowing the exploit to proceed as above [45]. Because this type of vulnerability is caused by faulty timing rather than out of bounds, we refer to use after free vulnerabilities as temporal memory violations.

2.1.3 Code Injection

The simplest method to obtain arbitrary code execution from a buffer overflow attack is code injection. In this attack, the attacker fills the buffer with machine code instructions, known as shellcode, and redirects the instruction pointer to execute the provided shellcode [34]. This led to modern systems enforcing that pages are never both writable and executable, a defense known as write xor execute ($W \oplus X$) or data execution prevention (DEP), and this has effectively defeated code injection attacks [49].

Figure 2-1: Stack layout of a return to libc attack.



2.1.4 Return-to-libc

Unfortunately, this defense is not sufficient because the attacker can still gain control of the stack and cause malicious behavior by executing only previously existing code; this class of attacks is called code reuse attacks. Usually, `libc` is targeted because of its varied access to system calls and its common inclusion in C programs. In a return-to-libc attack, an attacker overwrites the return address with the entry point of a function in `libc` such as `system` and additionally takes advantage of the calling convention where arguments are passed on the stack. This allows the attacker to execute the `system` function with an argument such as `"/bin/sh"`, giving the attacker a remote shell and leading to arbitrary code execution without code injection [48]. We provide an example stack layout in Figure 2-1.

2.1.5 Return Oriented Programming

In return oriented programming (ROP), the reused code is at the individual instruction granularity rather than that of entire functions [41]. This type of attack chains together short sequences of instructions followed by a return instruction, called gadgets, by corrupting the stack with multiple return addresses consisting of the entry points of each gadget. Thus, a return oriented programming exploit can be thought

of as a program with these gadgets as basic instructions, and the stack pointer assumes the role of the instruction pointer. For example, if we have the gadget `pop ebx; ret;`, then an attacker can use this gadget to place an arbitrary value into the register `ebx` since the attacker has full control over the stack, and when this gadget is combined with the gadget `mov eax, ebx;`, then the attacker obtains full control over register `eax` as well.

It has been shown that return oriented programming is Turing complete using the gadgets provided in `libc` and many other common libraries [41]. In practice, only a single function call is enough to cause malicious behavior, and often in practical attacks the main target is the function `mprotect`, which allows the attacker to remove (W $\oplus$ X) protection on injected code [39] [43].

2.1.6 Variations of Return Oriented Programming

Many variations of return oriented programming exist which use different gadget spaces than the original technique or are exploited differently, allowing them to bypass defenses against standard ROP. Jump oriented programming (JOP) uses gadgets that end in an indirect jump instruction rather than a return instruction and has been shown to be Turing complete as well [5]. Sigreturn oriented programming (SROP) requires only a single gadget corresponding to the end of the signal handing function, as this function restores a user context using data supplied by an attacker. Finally, counterfeit object-oriented programming (COOP) uses C++ virtual functions as gadgets and chains them by constructing an array of objects with carefully chosen vtable pointers [37].

2.1.7 Address Space Layout Randomization

Address space layout randomization (ASLR) randomizes the virtual addresses of the stack, heap, and code regions when loading an application. This makes it significantly more difficult for an attacker to reuse code because they cannot redirect the instruction pointer to the desired gadget addresses, often causing the program to crash. This

version of ASLR is implemented on modern systems and is necessary for any attack to bypass, though finer-grained implementations exist to provide additional security [46].

Unfortunately, this type of defense is bypassable if the gadget addresses can be leaked to the attacker. In standard ASLR, a single leaked address will reveal the randomized offset for the entire region, allowing an attacker to infer the locations of any desired gadget [3]. The simplest way to obtain such an address is a memory disclosure vulnerability such as a buffer overread, where a program reads data past the end of the buffer and discloses unintended information; one such example of this type of vulnerability is the recent Heartbleed bug [22].

2.1.8 Side-Channel Attacks

It is possible to convert buffer overflow vulnerabilities into disclosure vulnerabilities through side-channel attacks, where the attacker can infer information about the program without directly receiving it. Side-channel attacks can be classified into two types: fault analysis and timing [40]. In fault analysis attacks, information about the program's memory is disclosed by observing whether the program crashes.

One attack technique, Blind ROP [4], overwrites data on the stack one byte at a time and can infer that a byte is unimportant or unchanged if no crash occurs, so the value of all important bytes can be found. Once the return address is found on the stack, memory can be scanned by redirecting the return address and again observing the program's behavior. In this type of attack, it is necessary that the program restarts after a crash with the same randomized state, but this is often the case with applications such as web servers.

Another method to infer a byte's value is through timing: if the program contains a loop whose number of iterations corresponds to a dereferenced pointer, then an attacker can overflow the pointer and estimate the byte stored at that address by measuring the program's execution time, since the loop will execute more times if the byte has a larger value [40]. We have an example of such a loop in Figure 2-2. This type of attack requires the loop to have a measurable execution time and in general

Figure 2-2: A timing attack loop.

```
1 int n = 0;
2 while(n < *ptr) {
3         // loop body
4         n++;
5 }
```

will require more samples to reduce random variation, but is harder to detect because no crash occurs.

2.2 Defenses to Code Reuse Attacks

We now evaluate previously proposed defenses to code reuse attacks with respect to provided security and overhead. We say that a defense provides high security if no known attacks exist, medium security if attacks exist but require specific conditions for the attacker, and low security if any exploit can be modified to bypass the defense.

2.2.1 Code Diversification

Code diversification defenses randomly scramble the code so that it is more difficult for the attacker to find useful gadget locations. These defenses can range from scrambling basic blocks to randomizing individual instructions, and this is implemented through code rewriting at the compilation stage rather than determining a random value at runtime [24] [52] [35] [23]. Similar to ASLR, code diversification defenses are vulnerable to memory disclosure vulnerabilities, though an arbitrary read vulnerability is required rather than a single address leak because of the additional entropy. These defenses are also vulnerable to side-channel attacks because this allows buffer overflow vulnerabilities to be converted into disclosure vulnerabilities [40].

2.2.2 Memory Safety

Memory safety defenses prevent the memory violation vulnerabilities from occurring in the first place. There are two types of memory safety: spatial memory safety which prevents out-of-bounds accesses and temporal memory safety which prevents use after free accesses. Typically, spatial memory safety is implemented by storing base and bounds information for every memory allocation and these bounds are checked on every pointer dereference and updated when the pointer is updated. For temporal memory safety, every memory allocation must also store a unique identifier so that if the memory is freed and then reallocated to a different pointer, the identifier for that memory will change and this can be checked on dereference.

The best current implementation of memory safety is SoftBound + CETS, which uses a modified LLVM compiler to rewrite the source code to add the updates and checks [31] [32]. This implementation is one of the strongest defenses against code-reuse attacks, with the authors providing a Coq proof of security, and no bypasses currently exist. Unfortunately, this defense has a very high memory and runtime overhead because of the large metadata structure per pointer; the runtime overhead can be as high as 300%. This implementation is not currently used because C is typically used for low-level performance optimization; higher-level languages with built-in memory safety can be used instead if performance is not an issue.

We note that the authors have also proposed HardBound, a hardware-based implementation of spatial memory safety [15]. By adding specialized hardware to store and check base and bounds structures, the runtime overhead was reduced to slightly over 20%, though the memory overhead can still be as high as 200% in some cases. It is possible that the runtime overhead can be reduced further with a to-be-released extension by Intel, known as Memory Protection Extension (MPX), which adds support for bounds registers, bounds tables, and fast instructions which interact with these [25].

2.2.3 Heuristic Defenses

Heuristic-based defenses attempt to identify some aspect of normal program execution that is not present in typical ROP attacks and then add checks to ensure that this property is satisfied. The kBouncer [36] and ROPecker [8] defenses use the fact that the most useful gadgets tend to be short and that normal programs don't have as few instructions between calls and returns. They enforce this by using the Last Branch Record (LBR), which stores a history of the last 16 control flow transfers taken, and they enforce that the number of gadget-like transfers does not exceed some threshold.

While these defenses have very low overhead and do not require recompilation, they have been shown to be bypassable because the gadget space is still quite large. An attacker can flush the history of short gadgets by chaining long gadgets that do not alter the ROP program's state [6] [38] [14].

Another defense, ROPGuard, enforces that the stack pointer always points to the expected stack memory region and that this region can never become executable [19]. This part of ROPGuard is easy to bypass with an arbitrary write vulnerability and even the original authors mention that an attacker aware of ROPGuard can easily defeat it.

2.2.4 Control Flow Integrity

Control flow integrity (CFI) defenses enforce that control flow transfers like calls and returns always go to valid locations. If a function can only be called from a fixed set of locations, then that function must return to the instruction immediately following one of those call instructions [2]. In this way we can construct the control flow graph (CFG) of a program with the basic blocks of the program as nodes and add a directed edge from block A to block B if it is possible to transfer control flow from A to B.

The problem of constructing the exact CFG of a program is computationally difficult because indirect function calls can go anywhere in the program, so knowledge of the set of possible function pointer values is required. As a result, implementations of control flow integrity often utilize approximations of the CFG and allow control

flow transfers that are not intended, and these extra edges often lead to a bypass. The Data Structure Analysis (DSA) algorithm is the best scalable implementation of CFG construction [29] and we believe that even the graphs constructed by this algorithm may be too permissive. However, this defense can be made very efficient, with runtime overhead of at most 8.7% in the Forward Edge CFI implementation [47].

One technique which often augments CFI defenses is the shadow stack. Whenever a return address is pushed onto the stack, a copy of that return addresses is pushed onto a shadow stack somewhere else in memory, so if the original is ever corrupted then this can be detected on return [12]. The use of a shadow stack completely prevents all return oriented programming attacks, but other code pointers such as function pointers are not protected at all and thus this defense is vulnerable to the JOP and COOP variations of ROP [5] [37]. In addition, implementations of this defense are vulnerable to attacks which can corrupt the shadow stack structure in memory. The shadow stack does not generalize to other code pointer types because return addresses are never modified once they are stored onto the stack, unlike other code pointers, and it is difficult to determine whether a code pointer modification is intended.

In the most coarse-grained implementations of CFI, valid return addresses only need to follow a call instruction and call instructions must target the beginning of a function. The Compact Control Flow Integrity and Randomization (CCFIR) [55] and Control Flow Integrity for COTS Binaries (BinCFI) [56] defenses enforce both of these properties while ROPGuard [19] and kBouncer [36] enforce only the first. These constraints do reduce the gadget space for an attacker, but it has been shown that enough call-preceded gadgets still exist to make attacks possible [21] [38] [14]. The G-Free defense rewrites the code so that an attacker can only transfer control to function entry points [33], but this does not stop even return-to-libc attacks and arbitrary computation can still be obtained through the COOP attack [37]. In addition, G-Free attempts to enforce this by storing the return addresses xor a random key, so any memory disclosure will likely leak enough information such that an attacker will

be able to guess the random key.

The Cryptographically Enforced Control Flow Integrity (CCFI) defense attempts to protect all code pointers from corruption by storing an authentication code through encryption with a random key [30]. The key is stored in a secure register not used in the code, so unless an attacker can guess the key, the attacker cannot forge the authentication codes. This defense is not practical however because the encryption method used is only 1 round of AES, and this can be easily broken if an encrypted address is leaked. If more rounds of AES are used, the defense will become more secure but will then have too high of an overhead to be useful.

2.2.5 Code Pointer Integrity

Code Pointer Integrity (CPI) is a recent compiler-based defense which enforces memory safety only on code pointers [27]. Because code pointers only make up a small percentage of all pointers in a program, this reduces the overhead which makes memory safety defenses impractical. To protect the code pointers from being modified by non-code pointers, all code pointers are moved to a separate region of memory called the safe region.

CPI also offers safe stack and Code Pointer Separation (CPS) versions of CPI which increase efficiency at the cost of security. The safe stack implementation only separates code pointers stored on the stack to a new stack and the CPS implementation moves only code pointers to the safe region and does not enforce bounds checks.

Both safe stack and CPS are protected from traditional buffer overflows of non-code pointers on the stack because the code pointers are separated in a different region of memory and CPS is also protected from traditional overflows on the heap. However, neither defense protects against the case where a code pointer overflows into another code pointer and neither defense protects pointers to code pointers from overflows, as in the COOP attack [37]. CPI is not vulnerable to this type of attack but we will show that CPI is still vulnerable [17]. We perform a more detailed evaluation of CPI in chapter 3.

Chapter 3

Evaluating Code Pointer Integrity

In the previous chapter we summarized proposed defenses to code reuse attacks as well as their provided security guarantees and overhead. In this chapter, we take a closer look at the Code Pointer Integrity defense and present the result that relying on randomness to protect defense metadata is flawed [17]. We first describe the design and implementation of CPI and then demonstrate an attack against CPI's use of randomness to hide code pointers.

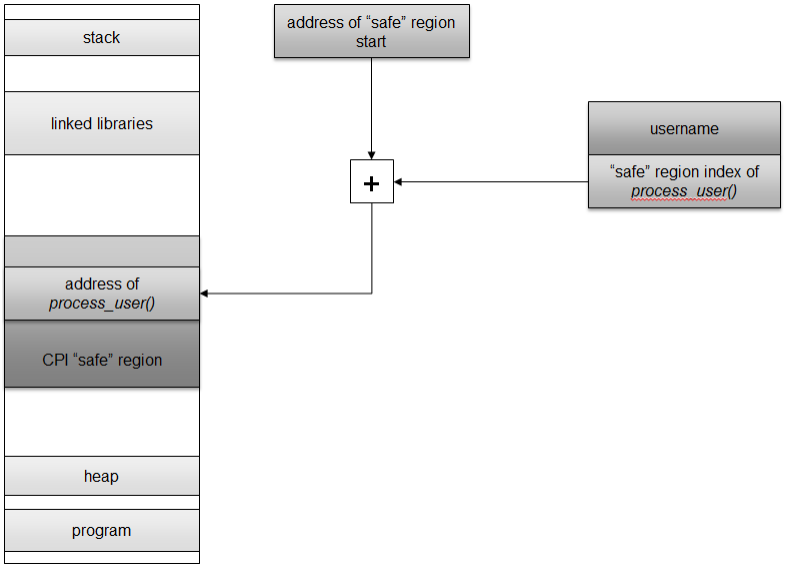
3.1 Code Pointer Integrity

Recall that CPI aims to protect code pointers by moving them to a separate region of memory, known as the safe region, and enforcing memory safety on this region only [27]. In order to do this, CPI first performs a static analysis to identify which pointers should be moved into the safe region, inserts code to perform bounds checks and access the safe region, and protects the safe region from being accessed by data pointers.

3.1.1 Static Analysis

CPI implements an LLVM pass which recursively identifies sensitive pointers as follows: all code pointers are sensitive and all pointers which may point to a sensitive

Figure 3-1: CPI safe region transformation.



pointer are sensitive. Pointers to sensitive pointers include pointers to structs which have a sensitive pointer as a component.

It is difficult to identify all code pointers in a program, so CPI tries to be conservative when guessing whether a pointer is a code pointer. In addition to function pointers, CPI also considers all `char*` and `void*` pointers as sensitive because these tend to be used as generic pointer types that function pointers can be cast from.

3.1.2 Instrumentation

Once the set of sensitive pointers is identified, their values are moved to the safe region and the old address is replaced by a unique identifier into the safe region. Figure 3-1 demonstrates this transformation: the true address of the function pointer `process_user` is moved to the safe region and the old pointer now contains its index into the safe region. CPI also stores the base and bounds for every pointer in the safe region in order to enforce memory safety.

If an attacker attempts to corrupt the offset and potentially overflow a different code pointer, CPI will detect this because every sensitive pointer dereference is checked against the base and bounds, so no pointer in the safe region can overflow

another pointer in the safe region. Currently, CPI only stores spatial memory safety metadata, but the structure can be easily updated to contain a unique identifier for each code pointer allocation to provide temporal memory safety as well.

CPI implements three types of safe region for mapping the identifier of a sensitive pointer to its location in the safe region. In the simple table implementation, CPI uses the key as a direct offset into the safe region. The lookup table implementation is similar, but it uses the key as an offset into an auxiliary table which then identifies the offset in the safe region. Finally, the hash table implements a hash map from key to sensitive pointer. The simple table implementation uses significantly more virtual memory space than the other two, but it is less expensive to maintain and requires fewer memory accesses to access sensitive pointers. We primarily evaluate the simple table implementation because it provides the best performance but show how our attack can be adapted to the other implementations as well.

3.1.3 Safe Region Isolation

On 32-bit systems, CPI relies on segmentation to isolate the safe region from other pointers. The safe region is in a different segment from the rest of the application and is thus inaccessible from non-code pointers. Segmentation is no longer supported on 64-bit systems, so CPI randomizes the base of the safe region. Because the virtual address space is so large, CPI assumes that an attacker would crash many times while trying to find the safe region due to referencing unmapped memory. While these crashes would cause the exploit to be detected, we will show that it is not necessarily the case that crashing is necessary to locate the safe region. In the next section, we show that CPI is vulnerable to side-channel disclosure attacks on 64-bit systems that allow an attacker to locate the safe region and bypass CPI.

3.2 Attack Methodology

3.2.1 Overview

Recall that CPI only protects code pointers from overflows; data pointer overflow vulnerabilities are still possible. If an attacker is able to determine the base address of the safe region, the attacker is then able to use a data pointer vulnerability to modify code pointers and bypass CPI.

In this section, we show that we are able to leak the safe region's address through fault analysis and timing side-channel attacks using a data pointer vulnerability. We also demonstrate a proof-of-concept exploit against the Nginx web server compiled with CPI and show that we are able to bypass CPI in 9 hours without any crashes and in 6 seconds with 13 crashes. We note that the original prototype of CPI released did not actually implement randomization and instead was mapped to a fixed address so we modified CPI to use a randomized mmap instead in order to attack the strongest implementation.

In order to bypass CPI, our attack proceeds in three phases. First, we must launch a timing side-channel attack by overflowing a data pointer which points to a value correlated to execution time. In our proof-of-concept exploit, the byte pointed to controls the number of iterations of a loop in the code. By overwriting the data pointer with an arbitrary address, we are able to leak information about any byte in memory. We must then calibrate the timing side-channel attack by gathering timing data. We do this by gathering a large number of response times for byte value 0 and byte value 255 and from these two values we can interpolate to estimate the value of any byte.

Second, we use the timing attack to scan memory for the beginning of the safe region. In this step we take advantage of the virtual memory layout of the target program and are able to guarantee a starting address for the scan that does not crash. We can then safely scan for the beginning of the safe region without crashing or we can scan more quickly at the cost of risking crashes. We use the timing attack to identify known signatures of bytes that allow us to determine our offset in the safe

Figure 3-2: Timing attack loop in `nginx_http_parse.c`.

```
for (i = 0; i < headers->nelts; i++)
        ...
```

region.

Finally, once we have identified the base of the safe region, we are then able to find the true address of any code pointer in the safe region. We then overwrite a function pointer to the beginning of a ROP chain and overwrite its base and bounds information so that CPI does not detect the memory violation.

3.2.2 Timing Side-Channel Attack

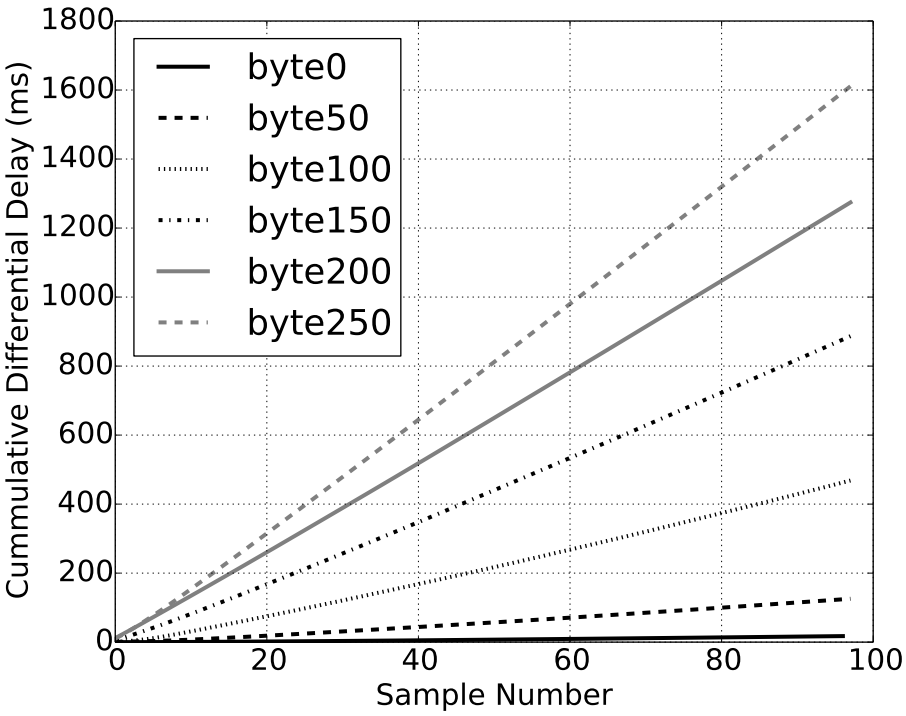
In order to launch a timing side-channel attack, we first require a data pointer vulnerability because CPI does not enforce memory safety bounds checks on these pointers. We can then overflow into a pointer used in a loop as in the following block of code. We see that the loop will iterate `*ptr` times, so if `ptr` points to a larger byte then the loop will take more time to execute. If the loop has a significant performance cost, then an attacker can measure this difference and estimate any byte in memory by overflowing `ptr` with any desired address. Figure 3-2 is the loop used by our attack: the pointer `headers` is dereferenced after adding a fixed offset, so the number of iterations in the loop satisfies our requirements for the timing attack.

We model the program's execution time as follows: we have a base execution time T_{base} which corresponds to the execution time of the program not counting the loop, we have the time per loop iteration T_{loop}, and we have a random term ϵ which corresponds to the random variance in the response time and has mean 0. If the byte value is x, then we obtain the following equation for the response time T:

$$T = T_{base} + T_{loop} \cdot x + \epsilon.$$

In order to reduce ϵ compared to the other terms, we only take the fastest 1% of samples, as recommended in [11]. We then sum over all n samples left, obtaining

Figure 3-3: Nginx timing measurements.



$$\sum_i T_i = n\left(T_{base} + T_{loop} \cdot x\right) + \sum_i \epsilon_i.$$

As n gets large, the RHS is dominated by the first term, so we can drop the error term. If we take samples at a byte where we know $x = 0$, then we can solve for T_{base} and obtain

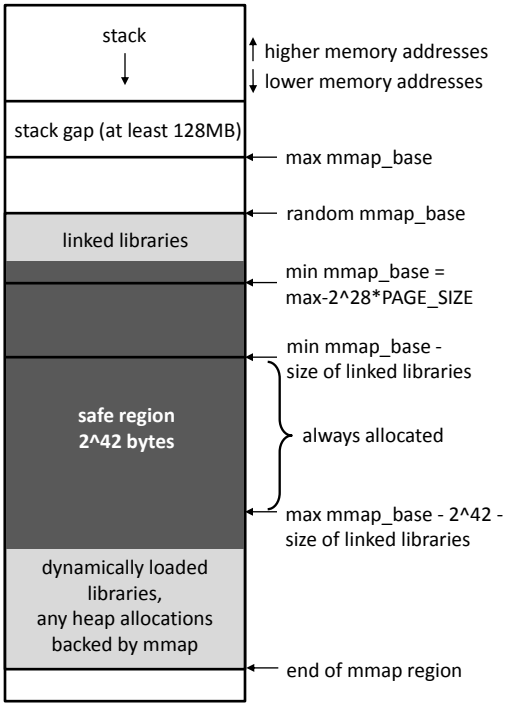
$$T_{base} = \frac{1}{n}\sum_i T_i.$$

We can then take samples at a known byte where $x = 255$ and solve for T_{loop}, obtaining

$$T_{loop} = \frac{1}{255}\left(\frac{1}{n}\sum_i T_i - T_{base}\right).$$

While any nonzero value of x would be sufficient, we choose $x = 255$ to maximize the cumulative delay relative to the error term as this is the maximum byte value, so we are able to obtain the most accurate estimate possible. Figure 3-3 shows our cumulative delay timing measurements which demonstrate dependence on byte value.

Figure 3-4: Virtual memory layout of a CPI-protected application.



More details regarding the timing attack and choice of parameters can be found in [40]. For the subsequent parts of the attack, we abstract this timing attack as a method for estimating the value of any byte in memory with some error threshold. We note that executing the timing attack was the most difficult part of the proof-of-concept exploit because it is difficult to get an accurate estimate when the loop's execution time is small.

3.2.3 Virtual Memory Layout

Once we have calibrated the timing attack, we will use it to determine the base address of the safe region. To do this, we must first analyze the virtual memory layout of the program and determine exactly how the safe region is mapped into memory.

The safe region is initialized very early in the program and is mapped into virtual memory through anonymous `mmap` immediately after the linked libraries are mapped in. Figure 3-4 shows the high address virtual memory layout for a program compiled with CPI on a 64-bit machine. We see that the largest possible `mmap`

base immediately follows the stack gap, and a randomized offset is subtracted from this max_mmap_base due to ASLR to obtain the true $mmap\_base$. This is where `mmap` begins allocating virtual addresses and does so contiguously, growing downwards. Within the `mmap` region starting at $mmap\_base$, the linked libraries are loaded first with the CPI safe region immediately following.

Note that the memory between the stack and $mmap\_base$ is not mapped and will cause a crash if dereferenced. In addition, each of the linked libraries loaded in allocates a 2MB library gap between the text and data segments. This gap has no read or write permissions and will cause a crash if dereferenced, so the library region is not entirely safe to dereference if we wish to avoid a crash.

3.2.4 Finding the Safe Region

The libraries loaded in are the same on many systems and we assume that the attacker knows this layout in advance with the exception of the randomized offset which determines $mmap\_base$ using the equation

$$mmap\_base = max\_mmap\_base - aslr\_offset.$$

The ASLR offset is measured in pages and has an entropy of 2^{28} in most Linux implementations, so with a standard page size of 2^{12} bytes, the maximum possible $aslr\_offset$ is 2^{40} bytes. In the simple table implementation, the safe region has a size of 2^{42} bytes which is larger than the maximum possible offset, so the address $max\_mmap\_base - 2^{40} - linked\_libraries\_size$ is guaranteed to be mapped and point to the safe region. However, the safe region is sparse and is mostly filled with zeros from the anonymous `mmap` initialization, so if we read a random byte from the safe region we will likely obtain no useful information about the offset.

We can however orient ourselves if we can read bytes from the linked library region, as the code segments are the same on many systems and contain unique byte sequences for identification. If we can obtain the offset of one of the libraries, this allows us to compute the ASLR offset and thus obtain the base address of the safe

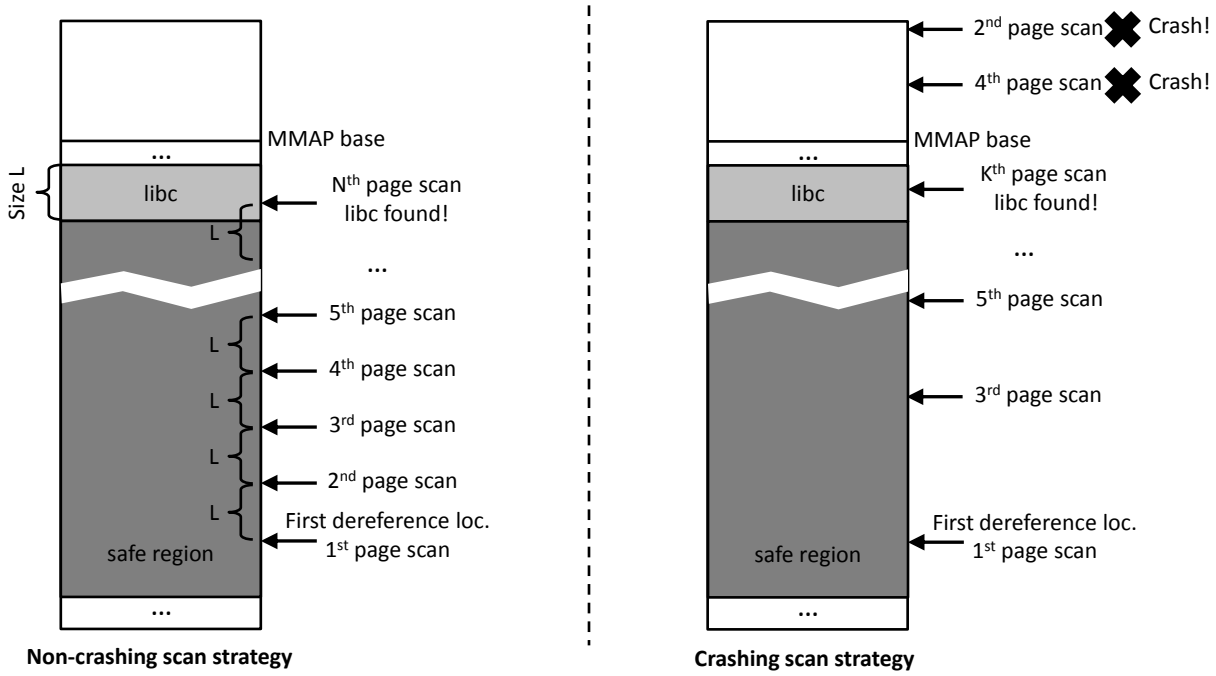
region.

We can thus perform the following attack to reach the linked library region: starting at the known safe address $max\_mmap\_base - 2^{40} - linked\_libraries\_size$, scan bytes going upwards until we reach the linked library region. While this attack will not crash, we will need to scan $\frac{1}{2} \cdot 2^{40} = 2^{39}$ bytes in expectation, and this is way too slow.

We can optimize this attack by noting that `mmap` is page-aligned, so rather than scanning individual bytes we can scan by multiples of the page size. In fact, we can do better than this because we don't necessarily need to scan the last page in the linked library region: any page will do, so we can scan by multiples of the linked library region size. Unfortunately, with this strategy we may crash if we deference one of the library gaps and there are some zero pages in the data segments, so this strategy doesn't quite work and the best we can do is by multiples of the size of the last segment loaded, which is the `libc` code segment. The size of this segment is approximately 2^{21} bytes, so our expected number of page scans is now $\frac{2^{39}}{2^{21}} = 2^{18}$, which is still slow, but will finish in a reasonable amount of time.

To determine whether a page belongs to the safe region or `libc`, we can carefully choose a small set of byte offsets to scan relative to the page base and sum their estimations. We have analyzed the individual pages in the `libc` code segment and determined that at least one of the bytes at offsets $(1248, 2176)$ is nonzero for every page in `libc`, so if the sum of estimations is nonzero then we have found a page in `libc`. The minimum possible sum of the bytes at these two offsets is 8, which is quite small, so we may obtain a false negative due to estimation error if we are unlucky enough to scan this page. It is better to instead use the triple $(704, 2912, 4000)$ which has minimum sum 56, since this makes it easier for the timing attack estimation to distinguish a `libc` page from a safe region page and not as many samples are required per byte to obtain enough accuracy. A false positive merely slows down our attack because we can detect this by scanning more bytes but a false negative would be disastrous. In this case, we would mistakenly identify the `libc` page as a safe region page and skip over this segment, causing a dereference of the `libc` library gap.

Figure 3-5: CPI attack strategies.



3.2.5 Finding the Safe Region with Crashes

We can reduce the number of page scans required to locate `libc` if we allow the attacker to crash some small number of times. Typically in web servers if a crash occurs then the parent process will restart the server without rerandomizing its address space. We can take advantage of this because if we guess that `libc` is located at address x, a crash tells us that our guess x is too high while no crash yields that x is too low, so we can perform a binary search. A naive binary search does not quite work, however, because it is possible for a guess to not crash if we guess a mapped address in the linked library region above the `libc` code segment.

To handle this case, we note that all of the other mapped library regions are very close to `libc`, so even if we do not locate `libc` exactly, we can find an address which is very close. When we execute the naive binary search, we always maintain the invariant that our high address causes a crash while our low address does not cause a crash. Figure 3-5 demonstrates the difference between the non-crashing strategy and the crashing strategy.

If we let $addr_{max}$ be the high address of the binary search interval and $addr_{min}$ be the low address, then it is guaranteed that $addr_{min} - linked\_libraries\_size$ is inside the safe region and its distance from the upper boundary of the safe region is at most $addr_{max} - addr_{min} + linked\_libraries\_size$. We can thus run the binary search until it is the case that $addr_{max} - addr_{min} \leq linked\_libraries\_size$ and then use the scanning strategy from the previous section beginning at address $addr_{min} - linked\_libraries\_size$. We also note that this scanning strategy can be done purely with fault analysis instead of the above timing method by scanning pages until a crash occurs.

In the binary search phase, we need to reduce a region of 2^{40} bytes to a region of $linked\_libraries\_size \approx 8 \cdot libc\_size \approx 2^{24}$ bytes. It follows that the number of binary search guesses required is $\log \frac{2^{40}}{2^{24}} = 16$, and in expectation $1/2$ of these will crash, for a total of 8 crashes in expectation. Afterwards, we scan from $addr_{min} - linked\_libraries\_size$ which is guaranteed to be at most distance

$$addr_{max} - addr_{min} + linked\_libraries\_size \leq 16 \cdot libc\_size$$

from the upper boundary of the safe region, so at most 16 more page scans are required, and these are guaranteed to not crash. In the proof-of-concept exploit, this stage of the attack required 12 crashes.

We note that midpoint binary search is not necessarily optimal for minimizing the number of crashes and provide a full analysis in [17].

3.2.6 Finding the Base Address of libc

Regardless of which strategy we used to scan for `libc`, we now have the address of a random page within `libc` and we need to determine the base address of `libc` while minimizing the number of bytes on the page that we will need to estimate. In the original attack as in [40], the library being leaked was randomized to the byte level, so the best the attacker could do was to scan a continuous sequence of bytes starting at the known address, but we can do better here because the library has only been

translated as a whole and because we are guaranteed that the randomized offset is page-aligned. The goal of this part of the attack is to identify a unique signature of bytes on the page that allows us to distinguish it from all of the other pages of `libc` in the presence of estimation error. Because we need greater accuracy in this part of the attack compared to checking if a page contains nonzero bytes, significantly many more samples are needed per byte in the timing attack, so it is important to scan as few as possible.

To handle estimation error, we will use the Hamming distance metric. Formally, if we use the timing attack to estimate the bytes at offsets $o_1, o_2, ..., o_k$ relative to the page base, then we obtain a length k vector x of estimates. We compare this vector to the reference vectors $r_1, r_2, ..., r_N$, where r_i is the length k vector constructed from the bytes at the same offsets using page i of `libc` and N is the number of pages in `libc`. We say that page i is a potential match if $dist(r_i, x) \leq D$, for some distance threshold parameter D. We wish to minimize k such that at most one of the r_i is a match. If none of the r_i match, then we have identified a false positive.

In our analysis of `libc`, we found that $N = 443$ and that the minimal Hamming distance between any two pages is 121, thus the maximal threshold D that will guarantee at most one match is $\frac{121-1}{2} = 60$. In our proof-of-concept exploit, we found that $D = 50$ was sufficient to handle any estimation error.

We now propose a greedy algorithm for determining choice of offsets. It is likely NP-hard to determine the optimal choice of offsets but we can still show that even in the worst case we will not need to scan many bytes. At each step of the algorithm, we maintain a set of candidate reference pages R which match all of the estimates we have seen so far. In order to determine which offset should be selected next, we pick the offset which maximizes the number of candidate pages we can eliminate from R when we obtain the worst possible estimate. Once we have chosen an offset, we use the timing attack to obtain an estimate at that offset and eliminate all pages in R that do not match at that offset. We provide pseudocode below:

function COUNT_ELIMS($offset, estimate, candidates$)

$score \leftarrow 0$

```
    for all page ∈ candidates do
        if |page[offset] − estimate| > D then
            score ← score + 1
        end if
    end for
    return score
end function

function NEXT_OFFSET(candidates)
    return arg max_offset min_estimate COUNT_ELIMS(offset, estimate, candidates)
end function

function FIND_LIBC
    R ← {page_1, page_2, ..., page_N}
    while |R| > 1 do
        offset ← NEXT_OFFSET(R)
        estimate ← TIMING_ATTACK(offset)
        for all page ∈ R do
            if |page[offset] − estimate| > D then
                R ← R − {page}
            end if
        end for
    end while
    return R
end function
```

To evaluate how many offsets we will need in the worst case, we would like to be able to test this algorithm with all possible sequences of estimates. We would however obtain a branching factor of 256 at each offset which quickly becomes infeasible to compute, so we apply approximations and get a slightly looser upper bound.

To reduce the branching factor, we choose a pivot b and distinguish between only

2 cases: either the estimate is at most b or it is greater than b. If we let R_1 be the set of pages in R that have byte value at most $b + D$ and R_2 be the set of pages in R that have byte value at least $b + 1 - D$, then no matter what we obtain as our estimate, the new candidate set will be a subset of either R_1 or R_2. It follows that if the worst case for R_1 takes at most n_1 offsets and the worst case for R_2 takes at most n_2 offsets, then the worst case for R will take at most $1 + \max(n_1, n_2)$ offsets. This lets us recurse on R_1, R_2 and obtain an upper bound on the number of offsets we will need to check. In our analysis we chose b to minimize $\max(|R_1|, |R_2|)$, similar to the greedy algorithm.

Using this method, we proved that for $D = 50$, at most 13 estimates will be required to uniquely identify the page regardless of what values for the estimates we obtain.

3.2.7 Finding the Base Address of libc with Crashes

If we allow crashes, then this problem becomes extremely trivial with a fault-analysis attack. Starting at our address in `libc`, we simply scan one page at a time until we obtain an address x such that x does not crash but $x + PAGE\_SIZE$ does. It follows that x must belong to the last page in `libc`, so we can obtain the base address of `libc` by subtracting off the known size of `libc`. This method requires exactly 1 crash and was used in the proof-of-concept exploit due to greatly speeding up the attack. Note that although there may be many page scans that do not crash, we only need one sample per address because there is no variance compared to the timing attack.

3.2.8 ROP Attack

Once we have determined the offset of `libc`, this allows us to compute the randomized `mmap` offset due to ASLR and this yields the base address of the safe region. At this point we have effectively bypassed CPI: we have completely determined CPI's hidden state for hiding the location of function pointers, so we can now determine

the new address of any function pointer in the program. We can thus overwrite a data pointer to point to a function pointer's address in the safe region and corrupt it to point to a ROP chain as well as corrupting the bounds structure so that CPI does not detect the corruption with a bounds check.

3.3 Discussion

While the version of CPI we attacked has several implementation flaws, they are difficult to patch and we believe that CPI's use of randomness to hide secrets in memory is fundamentally insecure.

One exploitable aspect of CPI was the use of contiguous `mmap` which ended up placing the safe region next to `libc`. Even if a non-contiguous `mmap` is used which places the safe region at a random address in the full address space, there still is not enough entropy and an attacker can brute force the location. In 64-bit systems, while there are 48 bits of address space, approximately only 46 are available for use. With a mapped region of size 2^{42}, the probability that an attacker guesses an address in the safe region is $\frac{2^{42}}{2^{46}} = 1/16$, so if the program does not rerandomize its address space, at most 16 guesses will be required to find an address in the safe region. We can then apply the binary search strategy to locate the edge of the safe region and obtain its base address, again bypassing CPI. We also note that non-contiguous `mmap` is discouraged in the official documentation: "The availability of a specific address range cannot be guaranteed, in general [1]."

To mitigate the brute force attack, we might try to use a smaller safe region size, as in the hash table implementation. We found that CPI required up to 2^{33} bytes for the hash table on the SPEC benchmarks, thus requiring at most $\frac{2^{46}}{2^{33}} = 2^{13}$ crashes to locate the safe region. This is still a weak security guarantee and we note that this number is close to the number of crashes required in the Blind ROP attack [4].

We can also consider an implementation of CPI that does not use a contiguous safe region, as in the lookup table. In this implementation, the safe region is broken up into subtables with a single master table which maps key to subtable index, and

each of these tables is mapped to a different region of memory. The total size of all subtables will need to be at least 2^{33} bytes to contain all sensitive pointers, so even in this case we can still leak a subtable address. Even if we do not know which pointers this subtable corresponds to, we can corrupt all of the pointers in this region to point to the ROP chain, similar to a heap spraying attack. We also note that this implementation of CPI does not work in the prototype and would likely introduce significant overhead due to the two-level lookup and lack of locality.

We note that since we have published our attack, the CPI team has updated their prototype with a Software Fault Isolation [51] based implementation [28]. This version of CPI no longer uses randomness to protect the safe region from other pointers and is thus not vulnerable to this attack.

Chapter 4

Tagged Architectures

The fundamental cause of code reuse attacks is that systems are unable to distinguish legitimate code pointers from attacker-corrupted ones, and the diversity of ROP-style attacks strongly suggests that an attacker-controlled code pointer is sufficient to obtain remote code execution. We thus believe that code reuse defenses should focus on protecting code pointers rather than mitigate potential exploits at a later stage. Ideally, we would like to be able to add a check on every pointer dereference to ensure that overflows do not occur in the first place, but as we have seen with SoftBound [31] the overhead is too high. However, if even one data pointer is unprotected, it can be used in an overflow, so we must assume that the attacker will be able to write to arbitrary memory.

CPI proposed the idea to instrument integrity metadata for only code pointers. The metadata would not be updated if a code pointer is corrupted and we would only have to check the metadata on code pointer dereferences, which occur significantly less frequently [27].

Unfortunately, current software-based defenses are unable to protect regions of memory containing defense metadata without significant overhead. If we do not add a check on every pointer dereference, then an attacker with an arbitrary write vulnerability can corrupt both the code pointer and the metadata, as in our attack on CPI [17], and thus bypass the integrity check.

In order to solve this problem, we propose modifying the hardware by adding tags

to every word of memory. These tags can be used to maintain information about the corresponding memory and can be protected from an attacker by making the tag memory unaddressable, so they cannot be corrupted. The tags can be updated in parallel with normal program execution, thus we can update and check tags with little overhead.

In this chapter, we first propose a secure method for verifying integrity of code pointers using a tagged architecture. We then compare to previous hardware-based defenses with the conclusion that our method is able to provide comparable security guarantees while significantly reducing memory overhead and increasing compatibility by reusing existing hardware.

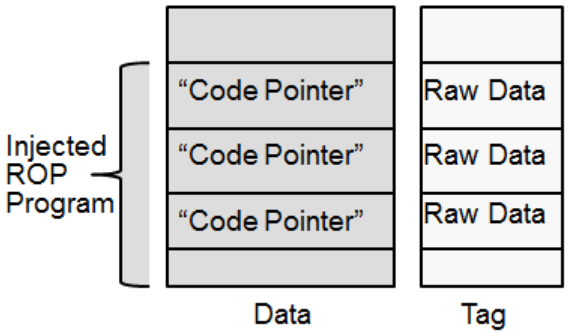
4.1 Design

4.1.1 Computation Model

In standard computation, we can abstract every instruction as a mapping. The inputs of this mapping consist of the input registers, the current program counter, and possibly a word of memory. The outputs consist of the value for the destination register or destination word of memory, the new program counter, and possibly a trap signal. For example, an addition instruction would define the output register value to be the sum of the two input register values.

In tagged architecture computation, the inputs now consist of tagged words with data and tag components rather than standard words. We thus define two mappings for every instruction: one for the data components, and one for the tag components. The data component mapping is the same as in standard computation, but the tag component mapping can vary based on what constraints are being enforced. We can think of this model as computing on the data component normally while updating information about the tags simultaneously. We refer to the tag component mapping as a policy.

Figure 4-1: Tagged Architecture Buffer Overflow.



4.1.2 Example Policy

Suppose that we have two types of tags: a return address (RA) tag which marks the associated word as a return address and a raw data tag for everything else. The only way to create a return address tag is from the call instruction, and the return address pushed onto the stack by the hardware will be marked with the RA tag. On a return instruction, the tag is checked and if the RA tag is not present the program will trap. In the case where the return address may need to be moved to a different address, the RA tag will be propagated on load and store instructions but all arithmetic will result in a raw data tag.

With this policy, if an attacker is able to corrupt memory from a buffer overflow, the injected ROP program will contain only raw data tags, as illustrated in Figure 4-1. On a return instruction, because the RA tag is not present, a trap will occur and the attacker will not be able to modify the control flow of the program.

4.1.3 Ideal Policy

Of course, it is not sufficient to protect only return addresses, and we must protect other code pointer types as well. Our ideal policy would provide two more tag types for function pointers and jump targets and check them on calls and jumps, respectively. We can propagate these tags similarly to return addresses except that we allow limited pointer arithmetic rules as follows: addition or subtraction of a code pointer and data yields a code pointer, but other types of arithmetic would clear the code pointer bit.

With this policy, it would be impossible for an attacker to forge a code pointer of any kind.

In fact, we can consider an even stronger policy which provides a tag bit for all pointers, not just code pointers, thus preventing all pointers from being forged. This would prevent attacks which corrupt pointers to code pointers such as the COOP attack [37] and make it impossible for an attacker to modify arbitrary memory, since the attacker would not be able to corrupt a pointer to point to the desired location to be overwritten. While it is still possible for an attacker to corrupt data local to an overflow, code reuse attacks are entirely prevented.

The advantage of these two policies over previous schemes is that both policies require very few tag bits per word. Thus, our approach not only has a very low memory overhead but also pays a smaller performance cost in caching tags due to the larger ratio of cache line size to tag size.

Unfortunately, both policies require compiler support because the hardware cannot initialize the correct tags, and compiler support is currently beyond the scope of our project. In the compilation stage, the information regarding which words correspond to pointers is lost, so both of these policies are more difficult to implement than the initial return address policy.

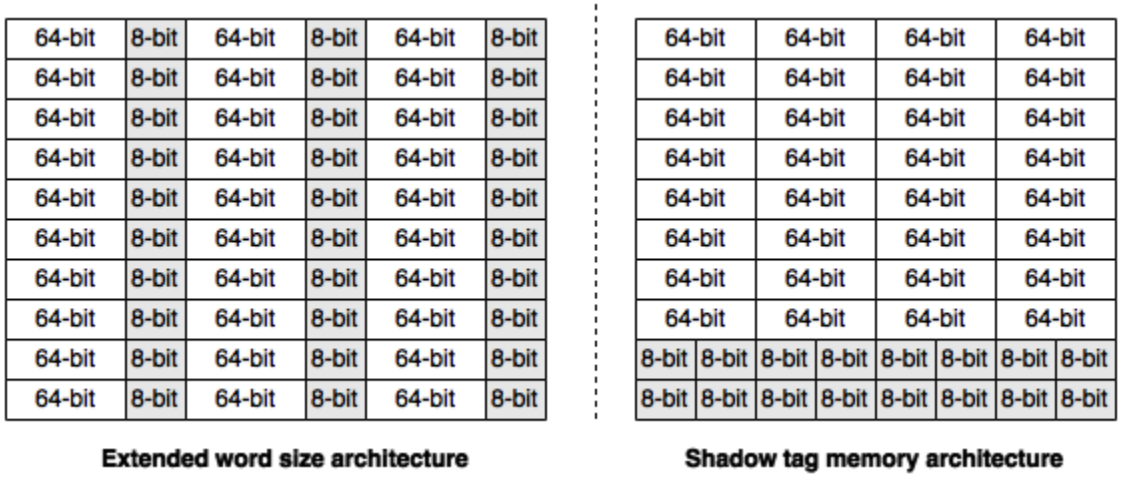
4.2 Our Approach

We now show how we can implement our tagged architecture model by reusing existing hardware components with low overhead.

4.2.1 Extending Memory

The simplest way to add tags to memory is to extend the word size. In this approach, the word size is incremented by the tag size and the tags are stored alongside the original value. For example, if we wish to extend a 64-bit word with an 8-bit tag, we extend the word size to 72 bits and treat the first 64 bits as the value with the last 8 bits as the tag. With this method, it is easy to perform operations on both the word

Figure 4-2: Tagged architecture memory layouts.



and tag simultaneously because they are adjacent in memory.

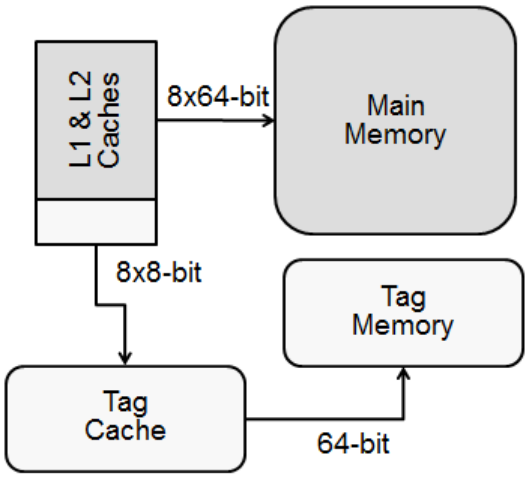
Unfortunately, this method is not practical because too many components would need to be changed to support the extended word size. We can mitigate this by instead allocating a region of shadow memory to store the tags while extending only the register word size. When a word is accessed from memory a second access for the tag becomes necessary, so this approach is not as efficient as extending the word size, though we will show how this can be mitigated with caching. The tag memory still has the original word size with each word containing several tags, so we do not have to extend the word size for the entire system and can reuse existing components. Figure 4-2 demonstrates the two memory layouts for a 64-bit word augmented with an 8-bit tag.

4.2.2 Tag Retrieval

In a shadow memory architecture, a naive implementation requires an extra memory access to retrieve tags. To mitigate this, we implement word-extended L1 and L2 caches and in addition introduce a tag cache as a miss handler for the L2 cache. With this cache architecture, an L1 or L2 cache hit would not require an extra memory access. Figure 4-3 illustrates our changes to the cache hierarchy.

If we miss in both L1 and L2 caches, we will need to go to main memory and the

Figure 4-3: Tagged architecture cache hierarchy.



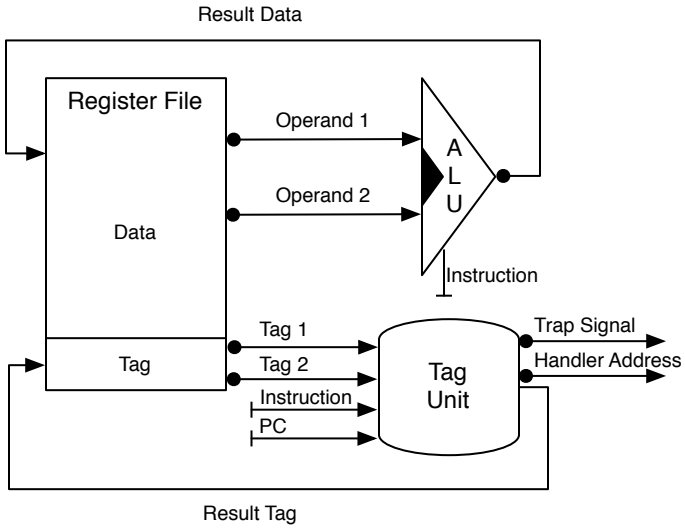
extra access is significantly slower, but we can sometimes avoid it with the tag cache. The tag cache only contains tags, so with a 64-bit word and an 8-bit tag, a cache line will contain a factor of 8 more tags than the other caches for the same amount of memory. Unfortunately, regardless of whatever cache hierarchy we use, we cannot avoid paying the cost of an extra memory access for compulsory misses, and we fully evaluate the cache overhead in section 6.3.

4.2.3 Tag Computation

To perform computations on the tags efficiently, we extend the register file to add tags and augment the ALU with a tag processing unit, which we illustrate in Figure 4-4. If the ALU takes input registers $r1$, $r2$, and potentially a value from memory m, the tag unit in parallel takes as input tag of $r1$, tag of $r2$, and maybe the tag on m and it computes an output tag for either the destination register or destination word of memory. The tag unit in addition may output a trap signal when a security violation is detected.

With this model, because the tag unit computation is done in parallel with the ALU, the overhead of computing the new tag is small. In addition, all tag unit computation is completely independent of the ALU, so this model will be compatible with existing binaries and there is no need for the tags to be addressable from memory,

Figure 4-4: Tag processing unit.



so even with powerful memory vulnerabilities an attacker cannot directly corrupt the tags.

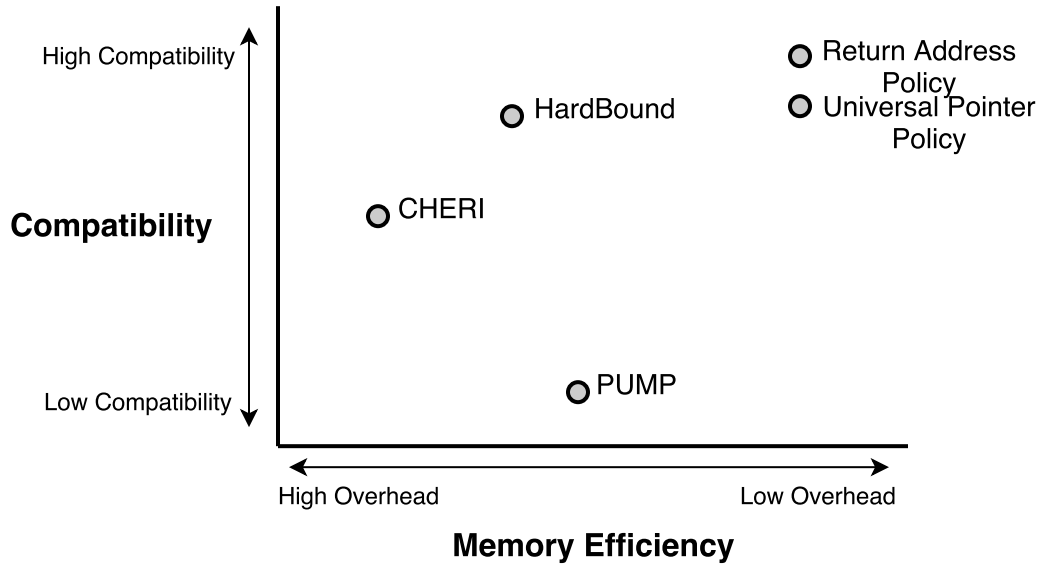
4.3 Previous Work

We now examine alternate hardware-based defenses against code reuse attacks and compare to our proposed policy. We evaluate each defense with respect to two metrics: memory overhead and compatibility, as all defenses presented here are believed to be secure. Figure 4-5 summarizes our comparison.

4.3.1 HardBound

The HardBound defense aims to enforce spatial memory safety using base and bounds pointer structures [15]. In this defense, every word of memory has a 1-bit tag which determines whether that word is a pointer. In addition, words that have this bit set also include base and bounds in a separate tag. This defense uses shadow tag space to store the tags with an L1 tag cache separate from both instruction and data. The tag updates are instrumented through compiler support which adds `setbounds` instructions and the checks are done purely in hardware.

Figure 4-5: Comparison of Hardware-Based Defenses.



This policy does provide slightly stronger security guarantees than our policy due to all overflows being prevented. However, the memory overhead for this policy is very significant, requiring almost 200 percent extra user pages on one test case. This policy is compatible with existing source code and does reuse existing hardware components, though lack of compatibility with binaries not compiled with the HardBound compiler is a slight downside.

4.3.2 CHERI

The Capability Hardware Enhanced RISC Instructions (CHERI) architecture is very similar to HardBound [9]. CHERI tags words of memory with 256-bit capability structures rather than base and bounds and uses the same 1-bit tag to mark which words have a corresponding capability. These capabilities enforce spatial and temporal memory safety in addition to finer grained control policies. While CHERI enforces stronger guarantees than HardBound, it suffers from the same compatibility and memory overhead issues.

One of the contributions of [9] is to classify idioms in C which cause issues for static

analysis such as storing pointers in ints or performing bitwise masking operations on pointers. These idioms are problematic for any tagging scheme which attempts to enforce type constraints.

4.3.3 PUMP

The Programmable Unit for Metadata Processing (PUMP) is very similar to our tagged architecture modifications in that both systems implement a method for enforcing custom tag policies [16]. The PUMP architecture augments every word with a 64-bit tag to support policies with an arbitrary number of tag bits, as tags can represent pointers to larger tag structures. A significant advantage of the PUMP architecture is that policies are programmed in software but enforced at the hardware level, allowing for customization.

Unfortunately, the PUMP has both high memory overhead and low compatibility. The 64-bit tag extension for every word of memory makes reusing existing hardware difficult and will likely impact cache performance as well if a shadow memory scheme is used. This tag also doubles the word size and thus incurs a large memory overhead of at least 100 percent, though the memory overhead may be larger if more than 64 tag bits are needed. We note that the PUMP architecture does do very well in terms of runtime overhead, with a 10 percent runtime overhead claim, so the idea of using a tagged architecture to enforce security is indeed possible to do efficiently.

Chapter 5

Taxi: A Minimal Secure Tagged Architecture

We now describe Taxi, our tagged architecture prototype on which we can evaluate our security policies. Taxi, tagged C, is a modified instruction set simulator for the open-source RISC-V architecture [54] which we have extended to support tags. In this chapter we primarily discuss our policies for implementing security and we evaluate them on our prototype in the next chapter.

5.1 RISC-V Architecture

We have chosen to extend the RISC-V architecture because it is open-source and because both software and hardware implementations are available, though RISC-V is still in its early stages [54]. In RISC-V, the instructions are 32-bit fixed-length, so a misaligned instruction pointer will not create unintended instructions. In addition, RISC-V is a reduced instruction set architecture, making it easier to modify and evaluate.

RISC-V provides an ISA simulator named spike [53], and this the primary component of Taxi. In addition, RISC-V provides a port of the Linux kernel and versions of both the `gcc` and `llvm` compilers, and we have modified these as well. Our full source code is available at `https://github.com/riscv-mit` [18].

Figure 5-1: Taxi Tagged Word Structure.

```
typedef uint64_t reg_t;
typedef uint8_t tag_t;
#define MEM_TO_TAG_RATIO (sizeof(reg_t) / sizeof(tag_t))

typedef struct {
  reg_t val;
  tag_t tag;
} tagged_reg_t;
```

Figure 5-2: Taxi Tag Memory Initialization.

```
mem = (char*) malloc(memsz);

tagsz = memsz / MEM_TO_TAG_RATIO;
tagmem = (char*) malloc(tagsz);
memset(tagmem, TAG_DEFAULT, tagsz);
```

5.1.1 Tag Extension

With a word length of 64 bits, we decided to implement an 8-bit tag because we wanted to be conservative with how many tag bits we would need and because this simplifies the implementation, as a byte is the smallest addressable unit of memory. To store the tags, we add a shadow memory region which simulates physical memory separate from the original memory region. This is illustrated in Figures 5-1 and 5-2, respectively. The shadow memory region is currently not addressable through virtual memory: we have considered adding mapping them to read-only virtual pages to give the operating system more control over the tags, but this may lead to aliasing concerns and we prefer the simpler implementation.

5.1.2 Arithmetic Instructions

To simulate the tag unit extension to the processor, we have implemented a set of macros that determine how tags are propagated for each instruction. For example, Figure 5-3 demonstrates our modified add instruction. We see that the new value for the destination register `RD` is the sum of the two input register value states `RS1`

Figure 5-3: Taxi Add Instruction.

```
1 WRITE_RD_AND_TAG(RS1 + RS2, TAG_ADD(TAG_S1, TAG_S2));
```

and `RS2` and we see that the new tag is determined by applying the `TAG_ADD` macro to the two input register tag states `TAG_S1` and `TAG_S2`. For instructions that use encoded immediate values rather than a second input register, we also define corresponding macros, such as `TAG_ADD_IMMEDIATE(TAG_S1)`.

We can thus define different policies by applying different definitions of these tag propagation macros. In `tagpolicy.h`, we have grouped these macro definitions and separated them with `#ifdef`.

We have found it convenient to distinguish between three types of arithmetic instructions: add/subtract instructions, bitwise logic instructions, and other arithmetic instructions. This is because instructions in the same group tend to be used similarly with respect to pointer arithmetic: it is common to add numbers to pointers and occasionally construct pointers from bitwise logic, but it is very rare to multiply pointers by constants. For instructions that take an immediate, we almost always just preserve the tag because the instruction only has a single input tag. This is also important because RISC-V does not provide a separate `mov` instruction to copy values from one register to another, and this is implemented as `addi rd, rs1, 0`.

5.1.3 Memory Instructions

RISC-V provides only two classes of instructions that interact with main memory: load and store, and there are separate instructions to load/store a byte, halfword (2 bytes), word (4 bytes), and double word (8 bytes). We have found the instructions that store less than 8 bytes to be problematic in our tagged architecture, since we implement one tag per 8 bytes of memory. Our initial implementation rewrites the tag for the full 8 bytes with the register's tag in this case, though we allow for policies to apply transformations to the tag first.

Figure 5-4 illustrates our modifications to the mmu interface for the load dou-

Figure 5-4: Taxi Load/Store Instructions.

```
1  tagged_reg_t load_tagged_uint64(reg_t addr) {
2    tagged_reg_t r;
3    void* paddr = translate(addr);
4    r.val = *(uint64_t*) paddr;
5    void* tagaddr = paddr_to_tagaddr(paddr);
6    r.tag = *(tag_t*) tagaddr;
7    return r;
8  }
9
10 void store_tagged_uint64(reg_t addr, uint64_t val, tag_t tag) {
11   tagged_reg_t r;
12   void* paddr = translate(addr);
13   *(uint64_t*) paddr = val;
14   void* tagaddr = paddr_to_tagaddr(paddr);
15   *(tag_t*) tagaddr = tag;
16 }
17
18 void* paddr_to_tagaddr(void* paddr) {
19   uint64_t offset = ((uint64_t) paddr - (uint64_t) mem);
20   return (void*) ((uint64_t) tagmem + offset / MEM_TO_TAG_RATIO);
21 }
```

bleword and store doubleword instructions, though the functions are very similar for the other type sizes. We first translate the virtual address to a physical one and load/store the corresponding register value. After, we map the physical word address to its corresponding tag address in tag memory and load/store the corresponding tag.

5.1.4 Jump Instructions

RISC-V provides two jump instructions: jump and link (`jal`), and jump and link register (`jalr`). These instruction first place the next program counter in the destination register and then jump to a target address. In `jal`, the target address is the sum of the current PC and an immediate, while in `jalr`, the target address is read from the input register `RS1`. We can consider jump and link to be similar to a call instruction, but with the return address placed in a specified register rather than on the stack.

For these jump instructions, the destination register value always propagates from the program counter, so we define a special constant `TAG_PC` and set the tag of the destination register to be this constant.

It is often convenient to distinguish between jumps, returns, and calls, and we can do this because of how the compiler conventionally outputs instructions for each case. In the case of a jump or return, the current PC will not need to be saved. Thus, the destination register will always be `r0` which is a special RISC-V register that always equals 0. RISC-V also defines a dedicated return address register `ra` which stores the return address, so in the case of a call the link register will be equal to `ra`. To distinguish between a jump and a return, only the return will have the target register be equal to `ra`. We summarize these conventions in Table 5.1, where `rx` represents any of the other registers and `addr` represents an address constant.

Table 5.1: RISC-V Jump Instruction Conventions.

x86 Instruction	**RISC-V Equivalent**
`jmp addr`	`jal r0, addr`
`jmp rx`	`jalr r0, rx`
`ret`	`jalr r0, ra`
`call addr`	`jal ra, addr`
`call rx`	`jalr ra, rx`

5.1.5 Tag Instructions

We additionally add two new instructions to RISC-V to complete the tag support. The `tagenforce imm` instruction is used to turn tag enforcement mode on or off. The instruction contains a single immediate and sets a special processor flag to that value, and we use that processor flag to determine whether to trap or not when a tag violation is detected.

We also add a `settag rd, imm` instruction which sets the tag of the destination register to an immediate. It is occasionally necessary to set the tag on a certain register due to exceptions in the policy or due to policies which need tags set from the compiler.

We have considered adding a `gettag rd, r1` which sets the value of the destination register to the tag of the input register. This instruction would allow for the compiler to enforce tag constraints and may have useful debugging capabilities, but we have not yet found this instruction to be necessary.

5.2 Implemented Policies

5.2.1 Basic Return Address Policy

The goal of this policy is to prevent attackers from modifying return addresses. We use a single bit, which we will call the return address or RA bit, and this bit is equal to 1 if the corresponding word is a return address and 0 otherwise. Initially, all of memory has the RA bit cleared, and the only way to obtain a set RA bit is from the jump and link instruction. The bit is copied on move, load, and store instructions, but is cleared on any arithmetic instructions, and this is summarized in Table 5.2.

Table 5.2: Return Address Tag Bit Propagation.

Instruction Class	Example	Destination Register Tag
Jump and link	`jalr rd, r1`	tag(`rd`) = 1
Arithmetic	`add rd, r1, r2`	tag(`rd`) = 0
Move	`addi rd, r1, 0`	tag(`rd`) = tag(`r1`)
Load	`ld rd, [r1]`	tag(`rd`) = tag([`r1`])
Store	`sd [r1], r2`	tag([`r1`]) = tag(`r2`)

When we encounter a return, we check if the bit is equal to 1 and trap if tag enforcement is on and this is not the case. Figure 5-5 illustrates our changes to the jump and link register instruction. Line 2 checks that tag enforcement is on, line 3 checks if the RA bit is not set, and line 4 checks if this is a return by checking the register containing the address to jump to. Note that we also only trap if we are not in supervisor mode, since the kernel tends to frequently break the call/return discipline.

With this policy, it is impossible for an attacker to use traditional exploits which overflow a return address. Because the attacker-supplied values will not be traced

Figure 5-5: Return Address Policy Jump and Link Instruction.

```
1 reg_t oldpc = npc;
2 if (TAG_ENFORCE_ON &&
3      (!(TAG_S1 & TAG_PC)) &&
4      (insn.rs1() == RA) &&
5      (!IS_SUPERVISOR)) {
6      TAG_TRAP();
7 }
8 set_pc((RS1 + insn.i_imm()) & ~reg_t(1));
9 WRITE_RD_AND_TAG(oldpc, TAG_PC);
```

back to a PC tag, the overflow store will clear the return address tag bit, so a trap will occur if tag enforcement is on.

This policy does have several important holes due to the fact that only return addresses are protected. Of course, an attacker can still corrupt a function pointer or jump address, but this policy is vulnerable to replay attacks through data pointer corruption. Consider the program in Figure 5-6 which contains an attack mechanism initially presented in [7]. If `buf1` and `buf2` are stored in registers, then when `vuln_fn()` is called, an attacker can corrupt the saved stack context and modify the two pointers to point to arbitrary locations in memory. When we get to the `strncpy` call, the attacker can move data in memory to arbitrary locations. We refer to such a vulnerability as an arbitrary copy vulnerability.

With such a vulnerability, an attacker can corrupt a return address by copying a return address from a different location, since this would preserve the RA tag. Even worse, an attacker can create arbitrary return addresses as follows: first, overflow the return address to the desired value. Next, find a return address in memory that still has the RA tag and has a byte that matches with the corrupted value just written. Finally, copy only that byte into the corrupted return address. After the first step, the value is chosen by the attacker with a cleared RA tag. The third step however restores the RA tag without changing the other bytes of the fake return address. Our next policies address these holes.

Figure 5-6: Example Replay Attack Gadget.

```
1 char* buf1;
2 char* buf2;
3 ...
4 vuln_fn();
5 ...
6 strncpy(buf1, buf2, buflen);
```

5.2.2 No Return Copy Policy

We first address the capability of the attacker to copy other return addresses in memory by enforcing that there is only one copy of a given return address at a time. This policy obeys the same rules as the basic return address policy, but additional changes are necessary. On arithmetic and move instructions, in addition to the rules for the destination register as in the previous policy, we also clear the RA bit on all input registers. On store instructions, we also clear the RA tag on the register being stored, and load instructions similarly clear the RA tag on the memory location loaded from.

This policy fixes the replay attack vulnerability from the previous policy since the only return addresses available for copying from are the ones yet to be used: these must be callers of the current context and the most an attacker can do is return early. However, this policy does not obey the restrictions set in our tagged architecture model described in Chapter 4. This policy modifies the tags on input registers and on memory locations loaded from, though at least the extra memory access is not too expensive due to the tag already being in cache.

This policy is also riskier because it makes several assumptions regarding well-behaved programs. This policy assumes that a return address will never need to be loaded from memory unless it is being used to return to. This policy also assumes that every return address will need to be used only once. While these assumptions are not true in general, we have found that in practice false positive traps from this policy can be mitigated because we can enumerate all locations in library code where they occur. We investigate this in Section 6.2 where we evaluate the policies on real

programs.

We note that our original version of this policy cleared the RA bit on the destination register tag if it is not the return address register `ra`. In this version of the policy, all copy vulnerabilities would be useless because the RA bit would be cleared on the load. Unfortunately, we found that this version of the policy caused too many spurious traps because there were legitimate cases where return addresses needed to be moved to other locations.

5.2.3 No Partial Copy Policy

We now address the capability of the attacker to only copy a single byte of a return address to change the tag on any word in memory without altering the value. In this policy, we start with the rules outlined in the basic return address policy, not the no return copy policy. We additionally add the rule that whenever we store a byte, halfword, or word, we clear the RA bit on the stored tag. It is necessary even in the word case to clear the tag: the high 32 bits of many code addresses are likely the same, giving the attacker freedom to choose the lower 32 bits.

With this modification in place, only 8-byte copies of return addresses are possible, so an attacker would have to copy the full value in order to get a valid RA tag. On its own, this policy would only allow copying of other return addresses in memory, which is arguably secure enough, but we can obtain additional security by combining this policy with the no return copy policy to guarantee integrity of return addresses.

We note that this policy is necessary only because our system only implements one tag per 8 bytes of memory, and it would take too many bits for every byte to have its own tag. However, even if this was the case and we enforced that all 8 bytes of a return address had the RA tag, some variant of this policy would still be necessary because an attacker may be able to create a new return address by copying valid bytes one at a time.

5.2.4 Blacklist No Partial Copy Policy

Up until now, all policies have only protected return addresses, and we would like to be able to protect other code pointers as well. Unfortunately, in this architecture, it is not clear how the initial tags for other code pointer types should be initialized without compiler support; many function pointers for example begin as indistinguishable constants in the executable. It is thus very difficult for the hardware alone to determine which words of memory are intended function pointers without help from the compiler, but perhaps we can try a different approach and find words of memory that cannot be code pointers: namely those that result from partial store instructions.

The primary assumption made by this policy is that when we encounter a byte, halfword, or word store instruction, the memory stored to cannot contain an 8-byte value and thus cannot be interpreted as a 64-bit address. This policy introduces a new tag bit which we call the data bit, and this bit is used to mark a word of memory as not a pointer of any kind. Initially, all of memory will not have this bit set. When we encounter a load, store, or jump instruction, we check that the address does not have the data bit set and trap otherwise.

Figure 5-7 illustrates our changes to the store byte instruction where we store register `r2` to an address determined by register `r1`. First, we check if `r1` has the data bit set and trap if this is the case, since in this case `r1` would not be considered a valid address. Next, because we are only storing a single byte, we set the data bit on the tag and store it to memory.

We have found it necessary to occasionally suspend adding data tags due to single-byte pointer copy loops, which we describe further in the next chapter. To allow for this we add an additional processor bit `DATA_ENFORCE_ON` which is set by default and only set the data tag if this bit is active. This bit is set and cleared using the `tagenforce` instruction.

With regard to tag propagation, the data bit is unique in that it indicates a lack of privilege, so it may be more useful to consider how a cleared data bit propagates. If the data bit is not present, this indicates that we do not know if the corresponding

Figure 5-7: Data Tag Propagation for Store Byte Instruction.

```
1  reg_t tag = TAG_S2;
2  reg_t addr = RS1 + insn.s_imm();
3
4  if (TAG_ENFORCE_ON &&
5      ((TAG_S1 & TAG_DATA)) &&
6      (!IS_SUPERVISOR)) {
7      TAG_TRAP();
8  }
9
10 if(!DATA_ENFORCE_ON)
11   tag = TAG_DATA;
12 MMU.store_tagged_uint8(addr, RS2, tag);
```

word is a pointer or data value and we must guess conservatively so that we do not cause spurious traps. We summarize the tag propagation rules for this policy in Table 5.3.

Table 5.3: Data Tag Bit Propagation.

Instruction Class	**Example**	**Destination Register Tag**
Add Arith	`add rd, r1, r2`	tag(rd) = tag(r1) & tag(r2)
Logic Arith	`or rd, r1, r2`	tag(rd) = tag(r1) & tag(r2)
Other Arith	`mul rd, r1, r2`	tag(rd) = tag(r1) | tag(r2)
Immediate Arith	`addi rd, r1, 1`	tag(rd) = tag(r1)
Move	`addi rd, r1, 0`	tag(rd) = tag(r1)
Load	`ld rd, [r1]`	tag(rd) = tag([r1])
Full Store	`sd [r1], r2`	tag([r1]) = tag(r2)
Partial Store	`sb [r1], r2`	tag([r1]) = 1
Jump and link	`jalr rd, r1`	tag(rd) = 0

For add/subtract arithmetic instructions, in pointer arithmetic the sum of a pointer and data value is considered a pointer, so we must clear the tag on the destination register if one of the input registers has a cleared data tag. For bitwise logic arithmetic instructions, although it is not common, we must again be conservative and clear the tag on the destination register if one of the input registers has a cleared data tag.

For other arithmetic instructions such as multiply or divide, however, we cannot

obtain a pointer if one of the input registers has a data bit set, so in this case we would set the data bit on the destination register.

For arithmetic instructions that take an immediate, the instruction length is 32 bits, so the immediate cannot be interpreted as a 64-bit address and we simply propagate the tag on the input register. We also copy the tag to the destination in the case of move, load, and store instructions. Jump and link instructions clear the data tag on the destination register.

It is difficult to evaluate the exact amount of security obtained by this policy because we are blacklisting words of memory, and this is generally a weaker approach than whitelisting as in the return address policy. However, one important aspect of this policy is that it guards against buffer overflow vulnerabilities involving attacker-supplied strings. All bytes in strings are marked with the data tag, so it is impossible for an attacker to corrupt any type of code pointer. In fact, the attacker cannot even use copy vulnerabilities because non-code pointers are protected as well.

Of course, because this policy is the most broad, it is more susceptible to spurious traps. There are cases where pointers are copied or constructed one byte at a time, but this number of instances where this occurs is limited and we discuss this further in section 6.2 where we test this policy on real-world programs.

5.3 Policies Requiring Compiler Support

We now describe policy ideas we have not yet fully implemented but which can provide additional security guarantees. The main difficulty in these policies tends to be that they require compiler support to set up the initial tags, and this requires significantly more work due to the increased complexity.

5.3.1 Function Pointer Policy

The goal of this policy is to protect function pointers in a manner similar to return addresses by adding a function pointer (FP) tag bit. When the compiler first loads a function pointer constant into a register, the compiler can then add a `settag`

instruction to set the FP bit to mark the initial function pointers, and these are determined through static analysis. The tags can then be propagated using pointer arithmetic rules similar to those in the blacklist no partial copy policy. On a call variation of the jump and link instruction, we would enforce that the FP bit is present and trap otherwise.

We have attempted to implement this policy as an LLVM compiler pass, but our implementation is currently incomplete due to LLVM losing information about which constants correspond to function addresses. This policy would also contain the same replay attack vulnerability as in the return address policy, but the same approach will not work in this case because function pointer rules cannot be as strict.

An example of a function pointer replay attack is the COOP attack [37] which overwrites pointers to function pointers. Even if we protect those as well with a new tag, triply indirect function pointers would still be unprotected, and so on. One possible solution to this issue is to use a static analysis similar to CPI's sensitive pointer analysis and mark those with the function pointer tag, but it is difficult to determine this set of pointers exactly. It may be the case that protecting only a specific type of pointer is too difficult and it might just be easier to tag all pointers instead.

5.3.2 Read-Only Function Pointer Policy

One potential mitigation against function pointer replay attacks is to distinguish between dynamic function pointers and read-only tables of function pointers in memory. This policy would introduce an extra bit to mark the second type of function pointer. This bit would obey the same rules as the function pointer policy except that on a store instruction, this bit would not be written back to memory. Thus, it would be impossible for an attacker to use a copy vulnerability to overwrite a dynamic function pointer with a read-only function pointer value. Again, we would require the compiler to insert `settag` instructions to initialize this tag bit.

However, pointers to function pointers are still unprotected and attacks using function pointers are still possible. In addition, attackers can still use copy vulner-

abilities using two dynamic function pointers, but this policy does at least remove the most useful function pointers from use by an attacker because the only the static function pointers have known values to the attacker.

5.3.3 Universal Pointer Policy

We can also consider a policy that maintains integrity of all pointers rather than only return addresses or function pointers by adding a pointer tag bit. This policy would need to be implemented similarly to the function pointer policy, though the static analysis would likely be easier. However, this policy would cause significantly more spurious traps due to strange C pointer idioms, such as those described in [9]. If this policy could be implemented perfectly, it would provide the strongest security guarantees by far: an attacker would be limited to corrupting nearby data variables only and this would stop almost all possible attacks.

5.4 Summary

We have implemented three policies which protect return addresses. The basic return address policy protects return addresses from overflows but is vulnerable to replay attacks where an attacker can forge the return address tag by copying from a different return address in memory. The no return copy policy aims to prevent this by clearing the return address tag on return addresses loaded from memory, preventing an attacker from reusing an old return address. Similarly, the no partial copy policy clears tags on byte, halfword, and word store instructions in order to prevent an attacker from forging a return pointer and then copying only a single byte from a different return address in order to add the return address tag to the corrupted address.

We have also implemented the no partial copy blacklist policy which is similar to the no partial copy policy but aims to protect all pointers rather than only return addresses. This policy goes one step further than clearing the return address tag on partial store instructions and instead marks that word of memory as not a pointer by setting the data tag bit. Although we cannot guarantee that this policy protects all

pointers, this policy does not require compiler support and thus is compatible with existing binaries. With compiler support, we would be able to implement policies which protect function pointers or all pointers in user applications and these would provide significantly stronger security guarantees than the four we have implemented.

Chapter 6

Evaluating Taxi

In the previous chapter, we have described our design for Taxi. In this chapter, we describe our prototype and evaluate policy compatibility and different tag cache sizes with real-world programs.

6.1 Prototype Components

6.1.1 ISA Simulator

As we briefly mentioned in the last chapter, the primary component of our prototype is a modified version of spike [53], a RISC-V ISA simulator. For each of the implemented policies, we have added a build script flag corresponding to that policy which is passed to the preprocessor and the code specific to that policy is surrounded by `#ifdef` blocks. The basic return address policy is always active, and at most one other policy can be active at a time.

6.1.2 Cross Compiler

Two cross compilers to RISC-V are currently available: gcc and LLVM. At the time of our testing, gcc is significantly more complete and is our compiler of choice. We have modified the assembler to add opcodes for the `settag` and `tagenforce` instructions. We have also modified glibc by adding these instructions to avoid spurious

traps, and we discuss this more in Section 7.2.

6.1.3 Linux

RISC-V also provides a port of the Linux kernel which can be cross-compiled and run inside spike, with the option to take a disk image file as an argument. We have added an extra hardware trap for tag violation traps and modified Linux to handle these traps by crashing the program.

We have also created several scripts to facilitate running programs inside the Linux kernel. Our primary script `setup_disk.sh` creates a blank disk image and populates it with the utility binary BusyBox [50]. This script also optionally copies in a directory passed by argument which contains cross-compiled binaries to test inside of Linux.

6.1.4 Cache Simulator

Spike includes a cache simulator which simulates a cache hierarchy. Although the simulated caches are not used in retrieving memory, they are nonetheless valuable for obtaining benchmarks on cache performance. We have modified the cache hierarchy to include a tag cache as a miss handler for the L2 cache and added support for multiple miss handlers so that we can evaluate different tag cache sizes on the same program execution.

This is implemented in spike through the memtracer API. Memtracer objects are notified whenever there is an access to memory, essentially duplicating every memory access.

6.1.5 Test Suite

Our test suite is designed to test for compatibility and prevention of exploits. The goal of our test suite is to verify that exploits are indeed prevented and to verify that programs that do not contain exploits execute normally without causing false tag traps. Unfortunately, we are not able to evaluate the performance overhead of

Figure 6-1: Trap Debug Mode.

```
#if TRAP_DEBUG
  #define TAG_TRAP() monitor()
#else
  #define TAG_TRAP() throw trap_tag_violation()
#endif
```

adding tags because an ISA simulator cannot measure the impact of having hardware components execute in parallel, but the cache simulator will let us test our overhead due to additional memory accesses. Recall that tags are not stored in the same region of physical memory as the rest of the corresponding word, thus requiring an extra access to main memory if we cannot retrieve the tag from cache.

We have written a number of single-file tests including both traditional exploits and theoretical exploits such as the replay attack discussed in the previous chapter. These programs also test signals, exceptions, and the problematic C idioms discussed in [9]. Our test suite also includes the Trinity system call fuzzer [26] and gcc torture tests [20]. In order to evaluate Taxi on real-world programs, we include the SPEC 2006 benchmarks [42].

6.1.6 Debugging

Spike includes an interactive mode similar to a gdb monitor which can be accessed by pressing Ctrl-C during execution, though much of gdb's tools were initially missing. In order to compensate, we have since added commands which print large regions of tagged memory as well as printing disassembled instructions. We have also augmented the debugger with a simple arithmetic parser which can also parse register names.

In `decode.h`, we define the preprocessor flag `TRAP_DEBUG` which enables debug mode if nonzero. In debug mode, tag traps are replaced with breaks to the debugger console, which we illustrate in Figure 6-1.

6.1.7 Libspike

We have found it useful for programs inside spike to communicate with spike. For example, a program may want to add a breakpoint to the spike monitor or to trace an address not known until runtime. We have thus mapped a special page of memory at the hardcoded virtual address `LIBSPIKE_BASE_ADDR` and handle accesses of this page differently in the MMU. Our interface allows programs to call functions in spike and receive data back through a union structure provided in `libspike.h`.

6.1.8 Memory Tracing

When debugging traps, often it is not very useful to view the program state when the trap occurs. This is because we are interested in determining when the tag was changed to its invalid value, not when the invalid tag was checked. Essentially, we would like to trace the history of a particular register value and find the instruction which wrote the bad tag.

We thus have implemented a memory tracing mode capable of maintaining this history. This mode is disabled by default due to overhead but can be enabled by passing the `-k` flag to our ISA simulator. If tracing is enabled, we can view the history through either the debugging console or through libspike.

Our memory tracing mode is implemented through node structures, illustrated in Figure 6-2. The basic idea is that whenever a register `rd` is updated with dependencies on input registers `r1`, `r2`, then we create a new node for `rd` pointing to the current nodes for `r1`, `r2`. We additionally add metadata concerning the instruction which caused the update so that we can determine the history of a register by following the input pointers and reading the node metadata. We not only initialize nodes for each register: we also initialize one for every word of memory.

In the node structure, the pointers to the input nodes are `input1`, `input2`. We also have `insn` which contains the instruction which updated `rd` and `pc` which contains the virtual address of `insn` which can then be looked up in an objdump output. The `dst` field describes the register or memory location updated by the

Figure 6-2: Memory Tracing Node Structure.

```
1 typedef struct node_t {
2   node_t *input1, *input2;
3   insn_t insn;
4   uint64_t pc;
5   uint64_t dst;
6   int is_mem;
7   int refc;
8 } node_t;
```

instruction. It is slightly ambiguous: in the case of a store instruction, it is equal to the virtual address stored to, and for all other instructions, it is equal to the index of the destination register.

The `is_mem` field encodes whether this instruction is a load/store instruction or not for the purpose of history lookup. In arithmetic instructions such as addition, both input nodes are considered equal and must be traversed. However, for load and store instructions, one input register corresponds to the actual value being loaded/stored and the other input register corresponds to the memory address. Most of the time we are only interested in the value node rather than the address, though there are some cases such as table lookup where the history of the address register is more important.

The `refc` field has nothing to do with the instruction and is there only for the purposes of cleaning up old nodes by maintaining a reference count. When we update a node on some register or address in memory, we decrease the reference count of the previous node. If it reaches zero, we recursively decrease the counts of the input nodes and clean them if necessary before deallocating the node.

We have implemented several optimizations so that the history does not get too large. First, we ignore changes to the stack pointer register because they are not relevant. Second, we ignore nodes if the PC is equal to the PC of one of the input nodes. This optimization is to deal with loops: the same register often gets updated several times at the same instruction. Finally, if an instruction only has one input register and the input is the same as the destination, then we do not consider this a meaningful change and do not use this node.

6.2 Policy Evaluation

6.2.1 Methodology

We now evaluate each of our policies with respect to security and compatibility using our test suite. For each of our test programs we cross-compile a binary using our modified version of gcc, create a disk image containing the binary, and run the binary inside of the Linux kernel running in spike.

We would like to be able to run applications inside of Taxi without having to change their source code and without causing false traps. We have found that outside of the gcc torture tests, it is very rare for application code to directly cause traps. Rather, the application code tends to call common library functions which then cause traps, and after patching the libraries, the traps disappear.

6.2.2 Security

We evaluate a policy's security by looking at which exploits the policy prevents. Our test suite covers return address overflows (`RA_OVERFLOW`), function pointer overflows (`FP_OVERFLOW`), return address replay attacks which copy the whole return address (`RA_REPLAY`), partial return address replay attacks which attempt to change only the tag (`RA_PARTIAL_REPLAY`), and data pointer overflows (`PTR_OVERFLOW`). We summarize our security results in Table 6.1

Table 6.1: Overflow Test Summary.

Policy	`RA`	`RA_REPLAY`	`RA_PARTIAL`	`FP`	`PTR`
Basic Return Address	Trap	No Trap	No Trap	No Trap	No Trap
No Return Copy	Trap	Trap	No Trap	No Trap	No Trap
No Partial Copy	Trap	No Trap	Trap	No Trap	No Trap
Blacklist	Trap*	Trap*	Trap*	Trap*	Trap*

Blacklist No Partial Copy Policy

For this policy, whether a trap occurs is dependent on the type of overflow. If the attacker data is copied into the buffer in 8-byte words, then no trap will occur, but if the copy is done 1 byte at a time at any stage, then the data tag will be applied and a trap will occur. In order to test which types of overflows will cause traps, we have written a test program which attempts to overflow a function pointer using various library functions.

In particular, we have tested a 1-byte for loop as well as `memcpy`, `memmove`, `strcpy`, `strncpy`, `strcat`, `strncat`, `gets`, and `read` using a file as input. We overflow a buffer whose size is a multiple of 8 with a string whose size is also a multiple of 8 to ensure that 8-byte copying is used if the optimization is present.

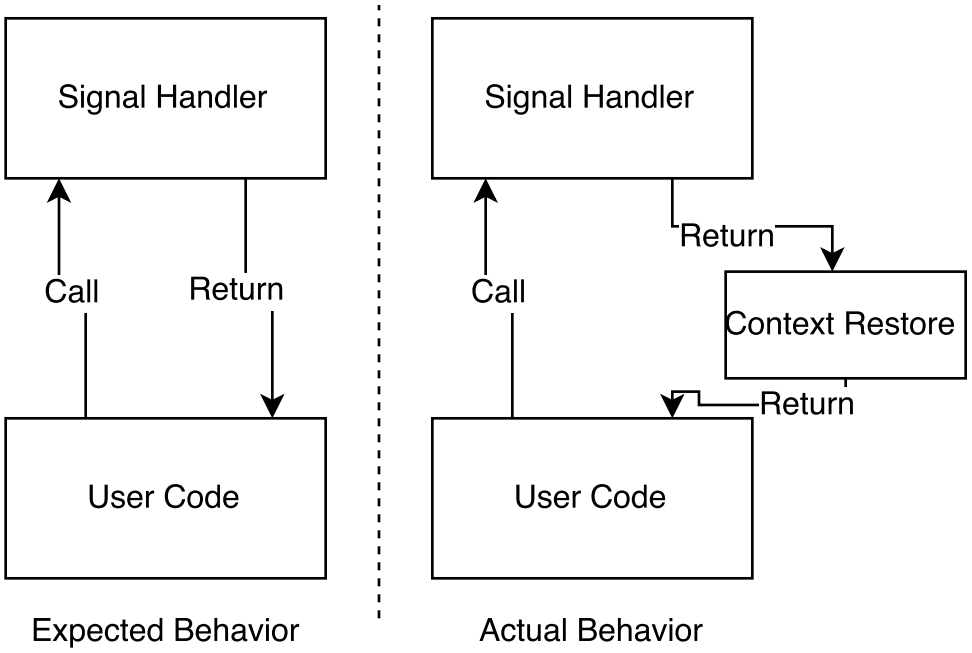
We have found that of these, only `memcpy`, `memmove`, and `read` did not cause a trap from the overflow. Of these, the first two are not supposed to clear tags because they are often called on structures containing function pointers, so this result is expected. What this means is that if an attacker-supplied string is ever passed through one of the other functions tested here, then the entire string will be marked as data and thus unusable in an exploit.

Unfortunately, our system does not currently support network sockets, so we were unable to test whether data read from the network would have the data tag. If this is the case, then web servers such as Apache and Nginx would be impossible to exploit remotely.

6.2.3 Return Address Policy

The return address policy is the most compatible: it enforces relatively lenient constraints on only return addresses and return addresses are relatively untouched by application code. As long as the call/return discipline is followed and return addresses are not modified, this policy will not trap. After applying the two modifications below to handle exceptions to the call/return discipline, this policy did not trap on any of our test programs.

Figure 6-3: Signal Handling Violation of Call-Return Discipline.



Signal Handling

When a signal is generated, the user context must be saved so that program execution can continue after the signal handler returns. The issue occurs when the user context is restored: the signal handler returns to trampoline code which restores the user context rather than returning to user code directly. The return to trampoline code is a new return address generated from the signal handler and thus is missing the return address tag, as illustrated in Figure 6-3.

To fix this, we modified the signal handler in the Linux kernel by adding the return address bit to the return address register immediately before returning. This is implemented through the instruction `settag ra, 1`.

We note that this instruction does not introduce a vulnerability because the return address that gains the return address tag is provided by the kernel, which we trust.

C++ Exceptions

C++ exceptions are implemented by pushing the address of the exception handler onto the stack and then returning to it. Because the return address was not created from a jump and link instruction, a tag trap occurs.

We prevented the trap by modifying the exception code to set the return address tag on the return address register before returning into the exception handler. We would have liked to implement this by modifying the `libgcc` source code, but this turned out to be too difficult because `libgcc` is a dependency of our cross compiler. We resolved this issue by manually patching the `libgcc_s.so.1` binary.

6.2.4 No Return Copy Policy

Recall that this policy augments the return address policy by enforcing that only one copy of the return address tag can exist at a time. With this constraint, replay attacks which copy return addresses should no longer work, and we have verified this in Taxi by constructing our own test exploit which traps under this policy but not the basic return address policy. Compatibility issues arise from this policy when a return address needs to be used multiple times or when a return address is loaded from memory in unrelated code.

After patching the issues below, this policy did not trap on any of our test programs except for the file `pr47237.c` in the gcc torture tests. This program tests the gcc `__builtin_apply` macro which provides a different interface for calling functions. The implementation of this macro scans the stack and loads the return address of `main`, which strips the tag on the stack copy and causes a trap when `main` returns.

Setjmp/Longjmp

The C standard library provides the functions `setjmp` and `longjmp` as primitives for implementing mechanisms which bypass the call/return discipline, such as exception handlers. In `setjmp`, the current register context is stored in a `jmp_buf` object.

Figure 6-4: Longjmp Modifications.

```
1 ENTRY (__longjmp)
2   REG_L ra,  0*SZREG(a0)
3   REG_L s0,  0*SZREG(a0)
4   settag s0, 1
5   REG_S s0,  0*SZREG(a0)
```

This context can then be restored through a call to `longjmp`, so it is as if the call to `longjmp` jumps to the instruction immediately after `setjmp`.

One component of the `jmp_buf` object is the return address of `setjmp`. In the initial call to `setjmp`, we need both the return address in the buffer and the return address on the stack to have the RA tag, but the policy enforces that only one copy of the tag can exist, so this policy would cause a trap. In addition, it is possible for a `jmp_buf` object to be used multiple times in a call to `longjmp`, and every call would require a copy of the tag.

The `setjmp` trap is easy to fix: we simply add a `settag` instruction which sets the RA tag on the `jmp_buf` copy of the return address. This will not introduce any vulnerabilities because we are only allowing a return address to be used twice rather than creating a new return address.

The `longjmp` trap is more difficult however because we must account for the possibility that an attacker has corrupted the stored return address in the `jmp_buf`, so we cannot use `settag` blindly. After loading the return address from the `jmp_buf` into the return address register, we then load another copy of the return address into a different register, call `settag` on that register, than store the return address back into the buffer, as illustrated in Figure 6-4.

If an attacker has corrupted the stored return address, then after the call to `longjmp` the corrupted return address will mistakenly have the return address tag but we will trap on return because the copy in `ra` will not, so this cannot be exploited. What we would actually like for this policy is a mechanism for copying tags or suspending the no return copy policy, but Taxi does not currently support this.

Fork

The `fork` function duplicates a process's state, including all saved return addresses in memory. Thus, if the tags are not also duplicated, then one of the two processes will trap. We prevent this trap by not applying this policy's rules to the kernel with the kernel propagating the tags according to the basic return policy rules.

Load/Store Instructions

In load/store instructions, the MMU will cause a page fault if the address being accessed is invalid in the page map. Our original implementation of this policy cleared the tag on the inputs before checking that the address was valid. The kernel would handle these faults by mapping in a new page and then attempt to execute the failed instruction again, resulting in the loss of the RA tag and causing spurious traps. This was fixed by first accessing the MMU before clearing the old tag.

6.2.5 No Partial Copy Policy

This policy caused no additional traps, though this is likely dependent on implementation. For example, we expected `memcpy` to cause traps due to return addresses being copied one byte at a time, but due to optimizations, `memcpy` actually only copied one byte at a time at the ends of the memory region and copied 8 bytes at a time whenever possible because this would require fewer instructions. However, `memmove` was not implemented the same way, but this was not an issue for return addresses because none of the code tested passed return addresses to `memmove`. As we will see in the next policy, this function does cause traps for other pointer types.

6.2.6 Blacklist No Partial Copy Policy

This policy is significantly more problematic than the previous return address policies because it attempts to protect all pointers rather than only return addresses. In addition, the edge cases for this policy exposed weaknesses in our original model that had not yet caused traps because of the rarity of return addresses.

False traps which occur due to this policy are caused when pointers are divided into bytes and then later recombined. This will likely be an issue for any theoretical policy, including the function pointer and universal pointer policies, since an attacker can use a single-byte copy vulnerability to combine a pointer using byte components of valid pointers. To distinguish valid pointer recombinations, we take the approach of whitelisting such code by suspending the policy.

With the below modifications to `memcpy` and `memmove`, this policy did not cause false traps on our test suite except for test case `960117-1.c` in the gcc torture tests and test cases `403.gcc`, `447.dealii`, and `483.xalancbmk` in the SPEC 2006 benchmarks. The latter two can be fixed through modification of `libstdc++`, which Taxi does not currently support.

Memcpy and Memmove

The functions `memcpy` and `memmove` are by far the most problematic for this policy because they occur frequently in application code and copy pointer structures one byte at a time. We thus modified these functions by adding a `tagenforce` instructions at function entry which suspends the policy, so the data tag is not added to the copied memory. We also add a second `tagenforce` instruction at the return of these functions which re-enables the policy so that the rest of the code is unaffected.

This applies to custom `memcpy` implementations as well, such as in test case `403.gcc` of the SPEC 2006 benchmarks. This application declares an in-place quicksort function `specqsort(base, n, size, compar)` which takes as input a base pointer to an array of items to be sorted, the number of items, the size of each item, and a comparator function pointer to compare them, respectively. When two items in the array need to be swapped in the sorting algorithm, this is done one byte at a time in a for loop which executes `size` times. This applies the data tag, so if `specqsort` is called on an array of pointers or even on an array of structs containing pointer components, a spurious trap will occur. This can be fixed by adding `tagenforce` instructions in the same way as above.

This problem also occurs in the `libstdc++` implementation of bitvectors in

Figure 6-5: Packed Struct Test Case.

```
1  typedef union T_VALS
2  {
3    char   *id __attribute__ ((aligned (2), packed)) ;
4  } VALS;
5
6  typedef struct T_VAL
7  {
8    short    pos __attribute__ ((aligned (2), packed)) ;
9    VALS    vals __attribute__ ((aligned (2), packed)) ;
10 } VAL;
```

`stl_bvector.h` when copying iterators and this causes traps in test cases `447.dealii` and `483.xalancbmk` of the SPEC 2006 benchmarks. This is not patched because the framework for modifying `libstdc++` has not yet been implemented.

Unaligned Packed Structs

The test case `960117-1.c` in the gcc torture tests declares a struct presented in Figure 6-5. We see that the `VAL` struct contains two components: a 2-byte short followed by an 8-byte char pointer. Because this struct is declared to be packed and aligned to 2-byte boundaries, the pointer is not aligned to an 8-byte boundary and shares the same 8-byte word as the short, with 2 bytes of the pointer overflowing into the next 8-byte word.

This is problematic both for this policy and for our original model. The primary issue here is that a pointer is sharing an 8-byte word with a data variable and thus is also sharing the tag, so it is impossible to accurately tag this word of memory. With respect to this policy specifically, when the short is assigned a value, the store halfword instruction will apply the data tag to the entire word, causing a trap when the pointer is dereferenced.

Thankfully, these unaligned packed structs are rare in both library and application code and we have not found any other traps of this nature. While we can mitigate this issue by adding a tag per byte rather than a tag per 8 bytes, we do not believe that the extra memory cost is worth the marginal compatibility gain.

6.2.7 Conclusion

We conclude that Taxi is indeed able to feasibly run real-world applications without causing false traps and with minimal source code modifications using these four policies. With all library modifications in place, only the heavily optimized gcc code caused traps out of all application code in our test suite. Taxi would definitely benefit from compiler support however. One could imagine a version of RISC-V with a load data or store data instruction which would remove the ambiguity in this type of copy loops.

6.3 Cache Evaluation

6.3.1 Methodology

We now evaluate the viability of the shadow memory cache hierarchy initially presented in Section 4.3. The goal of this section is to fully evaluate the overhead due to accessing main memory for tags in the presence of word-extended L1, L2 caches with a dedicated tag cache as an L2 miss handler. Using Taxi's cache simulator briefly described in Section 6.1, we can measure the tradeoff between tag cache size and additional accesses on real-world programs by running Taxi on the SPEC 2006 benchmark programs [42]. All testing was done using the basic return address policy, though choice of policy is irrelevant.

If we access an address not currently in a cache, then we issue a read access to the miss handler for that address. On write accesses, we mark the cache line as dirty and only flush this to the miss handler when the cache line gets evicted, and we refer to this case as a writeback. If a cache is too small, it is possible that the number of accesses to its miss handler exceeds the number of accesses to the cache because each write access will read in a cache line from the miss handler and then write it back on eviction.

Our cache hierarchy consists of two L1 caches: an instruction cache and a data cache. Both caches use the L2 cache as a miss handler, and the tag cache is the miss

handler for the L2. We set the cache sizes as follows: 8KB (+ 1KB for tags) 4-way associative L1 and instruction caches with 64-byte block size, 256KB (+ 32KB for tags) 8-way associative L2 cache with 128-byte block size. We test a tag cache with 4-way associativity and 64-byte cache line size and vary the size to be a power of two from 8KB to 8MB, for 11 tag cache sizes in total. Cache line eviction is chosen randomly through a fast xorshift generator.

In order to ensure that we are measuring only the relevant cache statistics, we have wrapped each test case inside a small program which first resets all caches through libspike before forking a process which runs the SPEC test case. This ensures that cache statistics due to unrelated code such as booting up Linux do not interfere. When the test case process exits, this program prints the cache statistics through libspike and returns. We exclude benchmarks `434.zeusmp`, `453.povray`, and `471.omnetpp` because they fail to complete.

We define the overhead of a particular test case to be the ratio of (number of tag cache misses + tag cache writebacks) divided by (number of L2 cache misses + L2 cache writebacks). Here, the numerator describes how many times we access main memory to store or load a tag while the denominator describes the number of times we access main memory to store or load a word.

6.3.2 Results

Our full results for tag cache overhead are presented in Tables 6.2 and 6.3 at the end of this section. We present the average overhead over all SPEC 2006 benchmarks in Figure 6-6.

As we expect, as the cache size gets large the overhead converges to the percentage of compulsory misses, but this number is insignificant for most benchmarks. We can still get significant gains with a tag cache size similar to the 256KB L2 cache size; with a threshold overhead of 10 percent, only 512KB of tag cache is needed. We note that these results use a 1-byte tag per 8-byte word while our policies currently use at most 2 tag bits. With a 2-bit tag per 8-byte word, we could reduce tag cache size by a factor of 4.

Figure 6-6: Mean Tag Cache Overhead.

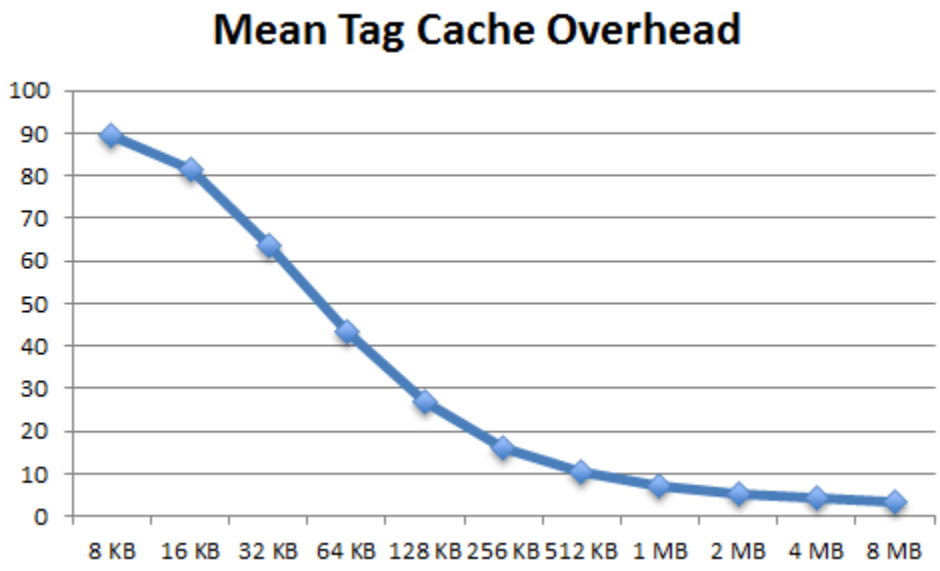
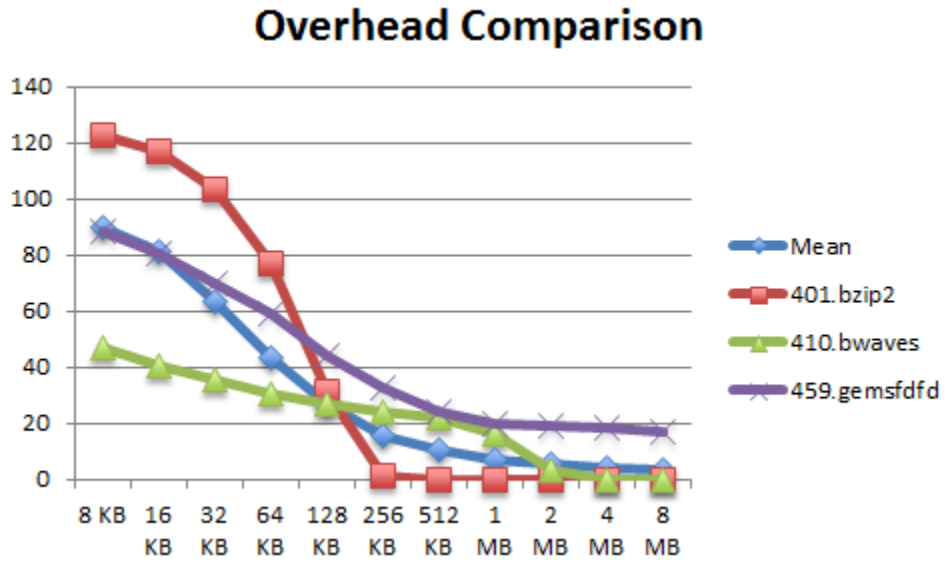


Figure 6-7: Tag Cache Overhead Comparison.



Only benchmarks `410.bwaves` and `459.gemsfdtd` exceeded 20 percent overhead with a 512KB cache. For benchmark `459.gemsfdtd`, even with an 8MB cache, we were only able to achieve 16 percent overhead, and this benchmark had one of the highest L2 miss rates at 35.9 percent. This suggests that mitigating the memory access overhead for this benchmark is likely impossible due to the large fraction of compulsory misses.

On the other hand, benchmark `410.bwaves` only had an L2 miss rate of 5.4 percent and for this program the ideal tag cache size is 2MB, with overhead 3.5 percent. This benchmark is unusual because of the large tag cache size needed to achieve a good overhead bound despite the low number of compulsory misses. In addition, this benchmark had the lowest miss rate with 8KB tag cache by far at 47 percent. We conjecture that this is due to the benchmark performing a numerical simulation on a 3 dimensional grid. If the entire grid can fit in cache, the number of evictions is greatly reduced.

Benchmarks `401.bzip2`, `433.milc`, and `473.astar` exhibit a different unusual pattern. These three benchmarks exceed 90 percent overhead with an 8KB tag cache and show very marginal improvement until a threshold tag cache size is reached which causes the overhead to drop significantly to below 1 percent. This is illustrated in Figure 6-7 with `401.bzip2` as the representative from this group.

We conclude that storing tags in shadow memory is practical. With a 512KB tag cache similar in size to the L2 cache, we achieve average overhead of 10 percent and can do significantly better on certain programs. The tag cache size can be improved with a smaller number of bits in the tag or with tag compression, though we have yet to investigate the impact on performance.

Table 6.2: Tag Cache Overhead Part 1.

	Tag Cache Size					
Bench	8 KB	16 KB	32 KB	64 KB	128 KB	256 KB
400.perlbench	89.32	81.06	58.66	32.65	13.47	5.52
401.bzip2	122.80	116.66	103.12	77.26	31.13	1.62
403.gcc	81.53	73.84	56.75	35.17	16.87	7.93
410.bwaves	46.99	40.51	35.27	30.50	26.83	24.12
416.gamess	88.69	80.01	60.11	38.54	21.18	12.75
429.mcf	86.14	79.87	64.67	39.98	18.95	10.10
433.milc	96.36	92.12	84.24	75.49	62.42	21.16
435.zeusmp	82.89	71.56	43.99	22.25	12.35	7.37
436.cactusadm	82.49	69.07	49.91	32.35	23.12	17.12
437.leslie3d	84.39	70.32	53.99	37.71	20.95	9.19
444.namd	78.28	69.96	51.84	33.55	21.00	13.82
445.gobmk	97.80	89.42	72.70	53.03	35.18	23.59
447.dealii	65.98	60.22	47.18	31.77	20.16	13.86
450.soplex	92.89	83.00	57.99	31.72	17.62	11.00
454.calculix	92.43	82.73	60.10	35.51	20.12	12.60
456.hmmer	91.63	82.47	60.39	37.55	21.51	13.13
458.sjeng	118.14	113.01	102.36	88.78	75.50	65.07
459.gemsfdtd	88.37	80.14	70.13	58.89	44.29	32.91
462.libquantum	90.26	81.88	62.78	41.03	23.96	14.41
464.h264ref	84.63	77.46	62.92	45.03	27.10	7.95
465.tonto	89.50	80.39	52.21	22.78	11.02	6.62
470.lbm	96.55	86.65	63.33	41.76	31.80	22.55
473.astar	133.70	126.45	106.72	69.46	23.25	1.50
481.wrf	79.55	69.84	50.77	31.59	21.21	15.62
482.sphinx3	62.24	56.40	48.02	40.08	34.35	27.99
483.xalancbmk	104.58	96.80	76.13	49.65	26.98	13.18
998.specrand	90.15	81.62	62.56	40.72	23.97	14.75
999.specrand	90.33	81.74	62.56	41.29	24.46	15.16

Table 6.3: Tag Cache Overhead Part 2.

	Tag Cache Size				
Bench	512 KB	1 MB	2 MB	4 MB	8 MB
400.perlbench	2.96	1.76	1.18	0.91	0.82
401.bzip2	0.14	0.03	0.01	0.01	0.00
403.gcc	2.85	0.50	0.28	0.23	0.22
410.bwaves	21.89	16.62	3.45	0.14	0.05
416.gamess	9.17	7.09	5.95	5.50	5.42
429.mcf	5.73	4.35	3.78	3.23	1.38
433.milc	0.20	0.00	0.00	0.00	0.00
435.zeusmp	3.48	1.81	1.30	1.06	0.99
436.cactusadm	12.78	1.20	0.29	0.21	0.19
437.leslie3d	1.55	0.01	0.00	0.00	0.00
444.namd	9.08	5.65	3.37	1.99	1.37
445.gobmk	15.66	7.45	0.38	0.05	0.04
447.dealii	8.59	3.32	1.91	0.60	0.24
450.soplex	7.90	6.59	6.09	5.96	5.87
454.calculix	8.97	7.16	6.47	6.18	6.08
456.hmmer	9.98	8.07	7.33	7.05	7.01
458.sjeng	57.02	49.75	40.96	28.80	12.05
459.gemsfdtd	24.24	20.06	19.17	18.37	16.82
462.libquantum	11.01	9.49	8.68	8.56	8.45
464.h264ref	1.13	0.23	0.08	0.05	0.04
465.tonto	4.84	3.77	3.24	3.03	2.97
470.lbm	14.41	11.06	9.62	9.59	9.28
473.astar	0.03	0.01	0.00	0.00	0.00
481.wrf	11.71	7.46	3.61	1.30	0.69
482.sphinx3	14.29	2.60	0.58	0.33	0.26
483.xalancbmk	5.22	3.43	2.73	2.44	2.41
998.specrand	11.21	9.49	8.79	8.62	8.53
999.specrand	11.39	9.65	8.99	8.59	8.53

Chapter 7

Future Work

Although Taxi is incomplete, we believe that tagged architectures are a promising defense mechanism against code reuse attacks. While current hardware-based defenses use too much memory or cannot reuse existing components, Taxi addresses both issues by using only 1 or 2 tag bits and minimally modifying existing processors and memory. In addition, Taxi provides strong security guarantees against code pointer corruption that cannot be matched by existing software defenses with the exception of full memory safety. We have demonstrated that almost all application code can run inside of Taxi without modification or causing false positive traps and that a hardware specification version of Taxi is practical under our model.

With compiler support, Taxi would be able to achieve even stronger security guarantees. Though Taxi supports complex tag propagation schemes, much of the type information present in source code is lost at the compilation stage and it is too difficult for the hardware to infer code pointer types on its own. With static analysis to identify function pointers and other pointer structures, we would be able to implement the ideal code pointer or universal pointer policies. A compiler which can distinguish tag copying instructions from tag clearing instructions would also be very helpful in eliminating compatibility issues in the no return copy and universal pointer blacklist policies.

One benchmark missing from our evaluation of Taxi is performance overhead due to tag computation. The full impact of parallelizing the tag unit with the ALU cannot

be measured in an ISA simulator and we would like to port our policies to a processor emulator such as QEMU or a hardware description language such as Bluespec or Chisel. Due to low overhead in the similar PUMP model [16], we are optimistic that our model is indeed practical. We have also not yet measured the impact of tag compression which would add a tradeoff between lower memory overhead at the cost of performance overhead. Our eventual goal is the development of a processor which we hope becomes widely adopted as a practical defense against code reuse attacks, and we believe that Taxi successfully demonstrates the viability of our model.

Bibliography

[1] mmap(3): map pages of memory - Linux man page. `http://linux.die.net/man/3/mmap`. 2015-07-15.

[2] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In Vijay Atluri, Catherine Meadows, and Ari Juels, editors, *ACM Conference on Computer and Communications Security*, pages 340–353. ACM, 2005.

[3] Anonymous. Bypassing Pax ASLR protection. *Phrack*, 11(59), July 2002.

[4] Andrea Bittau, Adam Belay, Ali Jose Mashtizadeh, David Mazieres, and Dan Boneh. Hacking blind. In *IEEE Symposium on Security and Privacy*, pages 227–242. IEEE Computer Society, 2014.

[5] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In Bruce S. N. Cheung, Lucas Chi Kwong Hui, Ravi S. Sandhu, and Duncan S. Wong, editors, *ASIACCS*, pages 30–40. ACM, 2011.

[6] Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security*, pages 385–399. USENIX Association, 2014.

[7] Shuo Chen, Jun Xu 0003, and Emre Can Sezer. Non-control-data attacks are realistic threats. In Patrick McDaniel, editor, *USENIX Security*. USENIX Association, 2005.

[8] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. Ropecker: A generic and practical approach for defending against rop attacks. In *NDSS*. The Internet Society, 2014.

[9] David Chisnall, Colin Rothwell, Robert N. M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. Beyond the pdp-11: Architectural support for a memory-safe c abstract machine. In ÃŰzcan ÃŰzturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *ASPLOS*, pages 117–130. ACM, 2015.

[10] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *SEC98*, 1998.

[11] Scott A. Crosby, Dan S. Wallach, and Rudolf H. Riedi. Opportunities and limits of remote timing attacks. *ACM Trans. Inf. Syst. Secur.*, 12(3), 2009.

[12] Thurston H. Y. Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn, editors, *ASIACCS*, pages 555–566. ACM, 2015.

[13] Mark Daniel, Jake Honoroff, and Charlie Miller. Engineering heap overflow exploits with javascript. In *Proceedings of the 2nd USENIX Workshop on Offensive Technologies*, 2008.

[14] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security*, pages 401–416. USENIX Association, 2014.

[15] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: architectural support for spatial safety of the c programming language. In Susan J. Eggers and James R. Larus, editors, *ASPLOS*, pages 103–114. ACM, 2008.

[16] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight Jr., Benjamin C. Pierce, and André DeHon. Architectural support for software-defined metadata processing. In ÃŰzcan ÃŰzturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *ASPLOS*, pages 487–502. ACM, 2015.

[17] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland'15)*, May 2015.

[18] Isaac Evans, Samuel Fingeret, and Julian Gonzalez. Risc-v mit project. `https://github.com/riscv-mit`, 2015.

[19] I. Fratric. Runtime prevention of return-oriented programming attacks. `http://ropguard.googlecode.com/svn-history/r2/trunk/doc/ropguard.pdf`. 2015-07-14.

[20] Free Software Foundation, Inc. GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF). `https://gcc.gnu.org/`. 2015-07-30.

[21] Enes Goktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy*, pages 575–589. IEEE Computer Society, 2014.

[22] Yongle Hao, Yizhen Jia, Baojiang Cui, Wei Xin, and Dehu Meng. Openssl heartbleed: Security management of implements of basic protocols. In *3PGCIC*, pages 520–524. IEEE, 2014.

[23] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. Ilr: Where'd my gadgets go? In *IEEE Symposium on Security and Privacy*, pages 571–585. IEEE Computer Society, 2012.

[24] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. Profile-guided automated software diversity. In *CGO*, pages 23:1–23:11. IEEE Computer Society, 2013.

[25] Intel. Introduction to Intel Memory Protection Extensions | Intel Developer Zone. `https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions`. 2015-07-14.

[26] Dave Jones. Trinity : A Linux system call fuzzer. `http://codemonkey.org.uk/projects/trinity/`. 2015-07-30.

[27] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In Jason Flinn and Hank Levy, editors, *OSDI*, pages 147–163. USENIX Association, 2014.

[28] Volodymyr Kuznetsov, LÃąszlÃş Szekeres, Mathias Payer, George Candea, and Dawn Song. Poster: Getting The Point (er): On the Feasibility of Attacks on Code-Pointer Integrity.

[29] Chris Lattner, Andrew Lenharth, and Vikram S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 278–289. ACM, 2007.

[30] Ali Jose Mashtizadeh, Andrea Bittau, David Mazieres, and Dan Boneh. Cryptographically enforced control flow integrity. *CoRR*, abs/1408.1451, 2014.

[31] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In Michael Hind and Amer Diwan, editors, *PLDI*, pages 245–258. ACM, 2009.

[32] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In Jan Vitek and Doug Lea, editors, *ISMM*, pages 31–40. ACM, 2010.

[33] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In Carrie Gates, Michael Franz, and John P. McDermott, editors, *ACSAC*, pages 49–58. ACM, 2010.

[34] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[35] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy*, pages 601–615. IEEE Computer Society, 2012.

[36] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In Samuel T. King, editor, *USENIX Security*, pages 447–462. USENIX Association, 2013.

[37] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *36th IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[38] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. Evaluating the effectiveness of current anti-rop defenses. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *RAID*, volume 8688 of *Lecture Notes in Computer Science*, pages 88–108. Springer, 2014.

[39] Offensive Security. Exploits Database by Offensive Security. `https://www.exploit-db.com/`. 2015-07-14.

[40] Jeff Seibert, Hamed Okkhravi, and Eric Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM Conference on Computer and Communications Security*, pages 54–65. ACM, 2014.

[41] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM Conference on Computer and Communications Security*, pages 552–561. ACM, 2007.

[42] Standard Performance Evaluation Corporation. SPEC CPU 2006. `https://www.spec.org/cpu2006/`. 2015-07-30.

[43] Corelan Team. Corelan ROPdb. `https://www.corelan.be/index.php/security/corelan-ropdb/`. 2015-07-14.

[44] Corelan Team. Exploit writing tutorial part 3 : SEH Based Exploits.

[45] Corelan Team. Exploiting CVE-2015-0311: A Use-After-Free in Adobe Flash Player | Core Security Blog.

[46] PaX Team. Pax aslr (address space layout randomization). `http://pax.grsecurity.net/docs/aslr.txt`. 2015-07-14.

[47] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Ulfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security*, pages 941–955. USENIX Association, 2014.

[48] Minh Tran, Mark Etheridge, Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *RAID*, volume 6961 of *Lecture Notes in Computer Science*, pages 121–141. Springer, 2011.

[49] Arjan van de Ven. New security enhancements in red hat enterprise linux v. 3, update 3. *Raleigh, North Carolina, USA: Red Hat*, 2004. 2015-07-14.

[50] Denys Vlasenko. Busybox. `http://www.busybox.net/`, 2015.

[51] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium of Operating System Principles*, 1993.

[52] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM Conference on Computer and Communications Security*, pages 157–168. ACM, 2012.

[53] Andrew Waterman and Yunsup Lee. Risc-v isa simulator. `https://github.com/riscv/riscv-isa-sim`, 2014.

[54] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. The risc-v instruction set manual, volume i: User-level isa, version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.

[55] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *IEEE Symposium on Security and Privacy*, pages 559–573. IEEE Computer Society, 2013.

[56] Mingwei Zhang and R. Sekar. Control flow integrity for cots binaries. In Samuel T. King, editor, *USENIX Security*, pages 337–352. USENIX Association, 2013.