Taxi: Defeating Code Reuse Attacks with Tagged Memory

by

Julián Armando González

S.B., 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 Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2015

Author .....
Department of Electrical Engineering and Computer Science
July 31, 2015

Certified by .....
Dr. Howard E. Shrobe
Principal Research Scientist, CSAIL
Thesis Supervisor

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

Taxi: Defeating Code Reuse Attacks with Tagged Memory

by

Julián Armando González

Submitted to the Department of Electrical Engineering and Computer Science
on July 31, 2015, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The rise of code reuse attacks has been devastating for users of languages like C and C++ that lack memory safety. We survey existing defenses to understand why none are generally applicable, focusing our attention on the Code Pointer Integrity (CPI) defense. We show that while CPI is hard to implement securely on modern architectures, it is based on the promising idea of storing metadata on memory. We also introduce Taxi (Tagged C), a set of hardware modifications that aim to prevent code reuse attacks by storing small amounts of memory metadata known as *tags* in hardware. Our reference implementation prevents several classes of code reuse attacks without losing compatibility with the C memory model and provides valuable insight into how tagged architectures can be used to enforce security properties on existing code.

Thesis Supervisor: Dr. Howard E. Shrobe
Title: Principal Research Scientist, CSAIL

Acknowledgments

I could not have performed any of this work without my wonderfully insightful and patient advisor Howard Shrobe, countless discussions and ideas from my labmate Sam Fingeret, and support from my family.

Finally, a sincere thanks to all that helped during research and read early drafts of this thesis, including Isaac Evans, Ulziibayar Otgonbaatar, Tiffany Tang, Stelios Sidiroglou-Douskos, Hamed Okhravi, and all of the summer UROP students. I have learned a great deal from you that I will never forget.

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 Tables

List of Figures

List of Source Code Listings

Chapter 1

Introduction

Memory corruption is a problem dating back at least to the 1970's [7]. The design of the popular C programming language makes it particularly vulnerable to attacks on memory, as it aims to provide programmers with expressiveness similar to assembly with as little overhead or management as possible. In line with this philosophy, C and similar languages like C++ do not provide memory safety, making programmers liable for much of their own memory management. In particular, programmers are responsible for preventing memory violations like buffer overflows [43].

This is not a responsibility that should be taken lightly. In C, introducing memory violations is easy as it only requires a single mistake. Consider Listing 1.1:

```
1 #include <stdio.h>
2
3 int main(void) {
4   char buf[64];
5
6   gets(buf);
7   printf("You said: %s\n", buf);
8   return 0;
9 }
```

Listing 1.1: A simple C program with a buffer overflow vulnerability.

This program uses the `gets` function to accept input from the user into a buffer and then outputs it to the console using `printf`. Despite its brevity, this program

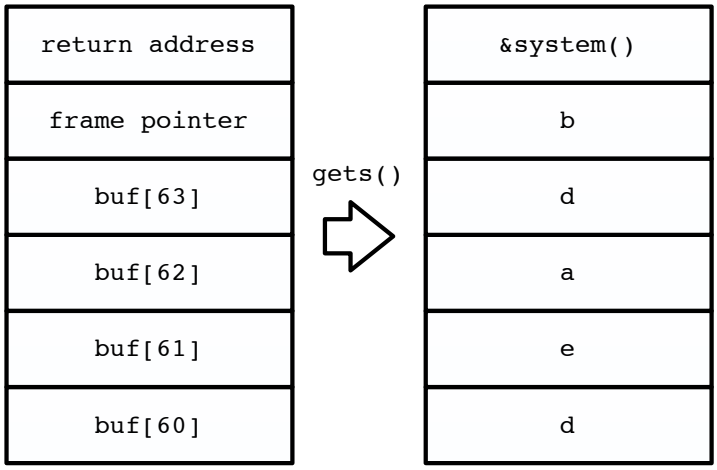


Figure 1-1: Diagram of stack before and after control flow redirection.

contains an easily exploitable buffer overflow: the `gets` function will accept input that is longer than the size of our buffer and happily overwrite memory adjacent to it. Because our buffer is stored on the stack (which may also hold the return address of the `main` function) this overflow can be utilized to redirect control flow. This situation is pictured in Figure 1-1.

The simplest way to fix this code is to replace the `gets` function with one that is aware of the buffer it's using, like `fgets`. Then, our simple C program may look like Listing 1.2.

```
1  #include <stdio.h>
2  #define BUFSIZE 64
3
4  int main(void) {
5    char buf[BUFSIZE];
6
7    fgets(buf, BUFSIZE, stdin);
8    printf("You said: %s\n", buf);
9    return 0;
10 }
```

Listing 1.2: Our simple C program but no longer vulnerable.

Using `fgets` neutralizes our buffer overflow at the cost of having to choose and keep track of the size of our buffer. C contains a number of commonly-used functions

that copy data into a buffer that also require knowledge of its size (like `snprintf`, `strncpy`, `memcpy`, and even system calls like `read`), and while this is not a problem for small programs, requiring programmers to correctly determine buffer sizes means that small mistakes can leave the program and possibly the entire system vulnerable.

A great example of such a vulnerability was the recent OpenSSL buffer overread known as Heartbleed [2]. This vulnerability, located in code parsing input from the network to handle the TLS heartbeat extension, relies on trusting the network to provide the length of its input. This length is then used as a buffer size argument for both a custom `malloc` and a custom `write` function. Because this length does not have to match the actual length of the input, malicious users can provide a length that allows reading up to 64 kilobytes of resident memory remotely.

As OpenSSL is widely used for providing SSL/TLS security throughout the world the impact of this bug was significant: major websites like Yahoo were affected, and many popular operating system distributions like Debian and Ubuntu were also vulnerable [45]. Some popular websites recommended that their users change their passwords following the vulnerability's disclosure. Like our simple C program, fixing Heartbleed was a straightforward task of inserting a few bounds checks. Nonetheless, despite OpenSSL's status as a well-maintained open-source project, the Heartbleed vulnerability was present in its source code for over two years before it was discovered.

Clearly, memory corruption attacks must be stopped. After an analysis of existing defenses, this thesis proposes a new hardware defense to a class of memory attacks called code reuse attacks. In Chapters 2 and 3 we explore the history of memory corruption attacks, tracing a path from buffer overflows to Return Oriented Programming (ROP) and other modern code reuse attacks. A diverse set of defenses have been proposed and implemented against these attacks, ranging from defenses that impose memory safety on C programs to heuristic-based defenses that try to prevent widely deployed exploits. All of these defenses require security, performance, and compatibility trade offs and none so far has proven generally applicable. Despite this fact, in Chapter 4 we identify the Code Pointer Integrity (CPI) [34] defense as a promising example of the role secure metadata can play in protecting memory.

In Chapter 5 we describe tagged architectures, which expand memory in hardware to include small amounts of metadata called *tags*. In Chapter 6 we introduce Taxi, a new hardware defense based on tagged memory. Taxi protects data that can regulate control flow like return addresses through minimal modifications to an existing architecture. By assuring the integrity of this data, Taxi deterministically detects and prevents many classes of code reuse attacks. In Chapter 7 we show the effectiveness of Taxi's guarantees on a modified RISC-V [8] instruction set simulator, demonstrating that tagged memory policies can be practical for many programs. We conclude in Chapter 8 with an outline of potential future work.

Chapter 2

Background

2.1 Early Memory Corruption Defenses

2.1.1 Stack Canaries

Some basic steps have been taken by developers of operating systems and compilers to counter the threat of buffer overflows. A very basic form of protection against buffer overflows on the stack is the use of canaries, or known values placed in memory that are checked before certain control flow transfers occur. In practice, canaries are often used on a program's stack to prevent return address corruption. In this case, a canary may rest between a buffer and a return address on the stack, and unless an attacker is able to overwrite the return address without changing the value of the canary a memory corruption attempt is detected and the program may exit.

The `gcc` compiler has had this protection available since version 4.1, which was released in 2005, although community versions of `gcc` providing this functionality have existed since at least 1997. Unfortunately, memory vulnerabilities often allow attackers to read stack canaries, negating their protection. They also do not prevent against buffer overflows that occur on the heap, and thus should not be considered a strong defense.

2.1.2 Non-executable Memory

The observation that programs executing code on the stack (or even the heap) can be an indication of successful code injection is one of the driving factors for the return of non-executable memory. The existence of separate memory for code and data is an architectural debate that goes back to the original Harvard and Von Neumann architectures, but was made possible at page granularity on the 64-bit version of the `x86` processor through the addition of the "NX" (AMD) or "XD" (Intel) bit to page table entries. An operating system with this bit available can configure memory like a user program's stack or heap regions to be non-executable. Additionally, this bit can be emulated in software, as many kernels for 32-bit `x86` processors without such hardware do [60]. In the context of preventing code injection, non-executable memory at page granularity is known as Write xor Execute (because it allows for memory that is not writable and executable) and as Data Execution Prevention (DEP).

DEP has become an effective tool against many attacks involving buffer overflows as it provides protection against most code injection attacks. Its hardware implementation also means that its protections come at a low space and performance cost: one bit of data for an entire page of memory that a hardware MMU can check efficiently. By raising the bar for attackers considerably, DEP has become one of the most effective defenses against memory corruption that currently exist.

2.1.3 Address Space Layout Randomization

With DEP in place, an attacker cannot use a buffer overflow to inject code into a program and run it. Attackers are not prevented however from directing a program to execute code already in its text segment or in a loaded library. This means that attackers can overwrite return addresses on the stack with addresses of powerful functions like the system call wrappers provided in the C runtime library, performing a return-to-`libc` attack. Address Space Layout Randomization (ASLR) aims to defend against this attack by randomizing on startup the base location of program sections like the stack and heap. Under ASLR, attackers wishing to use program addresses

must first determine them at runtime, often through guessing [61]. Some fine-grained variants even permute the order of individual functions [32].

ASLR, like DEP, significantly raises the bar for attackers trying to perform a memory corruption attack. Unfortunately, on 32-bit and narrower architectures ASLR does not provide strong security: many implementations provide low amounts of address entropy (commonly 16 bits) which can be brute-forced in a matter of minutes [50]. On 64-bit architectures, ASLR makes attacks more difficult but is still susceptible: a single memory disclosure vulnerability can reduce the work an attacker must perform to finding the offset of the desired function instead of its absolute address. In the presence of side-channels, memory disclosure vulnerabilities can also be manufactured out of existing buffer overflows [26]. Many implementations of ASLR also do not re-randomize memory locations when a process calls `fork`, which makes web servers that `fork` off worker processes especially vulnerable. Variants of ASLR with larger entropy by design and other code diversification defenses also can come with modest performance penalties, like Position-Independent Executables, which can require the use of a register to hold the program base address [58].

It is also notable that defenses like DEP and ASLR impose limitations on program design, like the avoidance of executing code on the stack or reliance on specific program addresses for functionality. While many programs do not use these features, DEP and ASLR conceptually provide safety by restricting a programmer's feature set, an approach that requires more than just changes to existing code: it requires modifications to the contract that languages and operating systems have established with long-existing programs. Like the responsibility to manage memory, these requirements create more opportunities for bugs, and must be weighed against the safety they provide. As an example, DEP complicates implementation of dynamic (Just-In-Time) compilers.

2.2 Code Reuse Attacks

Because DEP is such an effective and widely available solution for preventing code injection attacks, attackers have had to become creative with using code already present in a program binary or loaded library for malicious purposes. As mentioned in Section 2.1.2, DEP does not prevent so-called return-into-`libc` attacks where attackers call a powerful function in the C runtime library like a system call wrapper. As Shacham [49] notes, were this the only problem with DEP attackers would be limited to calling `libc` functions one at a time. This could make constructing attacks difficult if dangerous functions like `system`, which can be used to launch a shell with full system access, were removed. In this case, the maintainers of `libc` variants would face a large responsibility for the safety of C programs in general.

Return-into-`libc` attacks are not the only form of what can be called code reuse attacks. C and C++ do not require functions to be entered at their entry points: return addresses, function pointers, and other control flow data like C++ virtual table pointers are completely free to point to arbitrary instructions. A memory corruption attack that exploits this property is called a Return Oriented Programming (ROP) attack; there are many ROP variants.

2.2.1 Return Oriented Programming

As originally described by Shacham [49], ROP attacks rely on short series of instructions called "gadgets" that end in return instructions. Gadgets typically perform small tasks like pushing a value onto the stack or popping a value into a specific register and have very few side effects. On `x86` variants, attackers can execute gadgets sequentially by placing their addresses next to each other on the stack and using the `ret` instruction that ends each gadget to transfer control to the next gadget. The limits of attackers with access to the source code of a program then are only dictated by what instructions are present in the program and any loaded libraries. If we imagine gadgets as assembly in a strange computer, only a modest set of them is required for attackers to have a Turing-complete language.

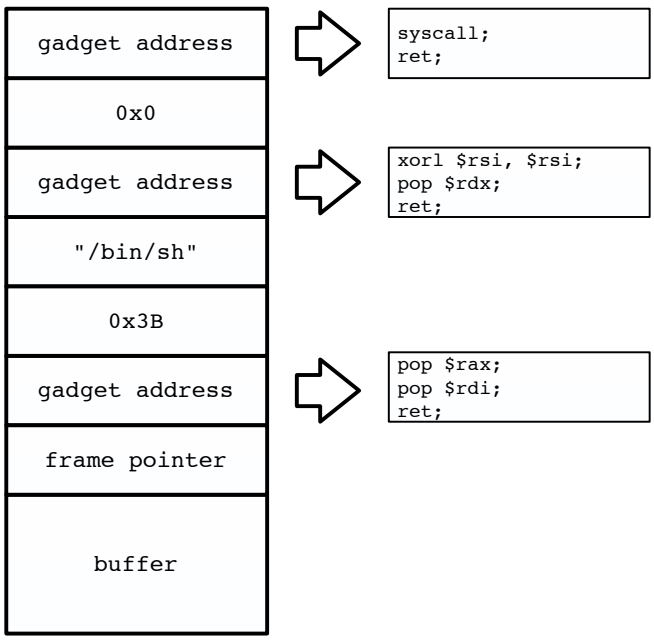


Figure 2-1: Simple ROP attack that makes a system call.

An example of a ROP attack that makes a system call to launch a shell is provided in Figure 2-1. In this figure, an attacker uses three gadgets. The first allows the attacker to load the system call number for `exec` into `$rax` and its first argument, a path to a shell, into `$rdi`. The second zeroes out `exec`'s other two arguments, while the third makes the actual system call.

ROP attacks are more expressive in architectures with variable-length instructions, as attackers are not necessarily limited to interpret a program's instruction stream as intended. In fact, the `x86` instruction set is extremely dense: even a random byte stream can be interpreted as a sequence of instructions with high probability [49].

As mentioned in Section 2.1.3, ASLR does not prevent attackers from using gadgets discovered through a memory disclosure vulnerability, when there is low address entropy, or after certain operations like `fork`. Because DEP and stack canaries only effectively prevent code injection, ROP attacks are very possible even with commonly-adopted security techniques.

2.2.2 Other Variants of ROP Attacks

Jump Oriented Programming

Code reuse attacks do not always require the use of return instructions. In Jump Oriented Programming (JOP), Bletsch, et al. describe a ROP variant that uses indirect jump instructions in place of return instructions [10]. JOP relies on a dispatcher gadget responsible for jumping to functional gadgets, taking the place of the stack in a traditional ROP attack. Instead of returning, each functional gadget jumps back to the dispatcher gadget, which then calculates the address of the next functional gadget. Because JOP does not depend on the stack for control flow, it can be more flexible than a ROP attack to perform with a heap buffer overflow. In short, JOP demonstrates that ROP-style attacks cannot be completely defended against by simply using compilers that avoid the use of return instructions: ROP attacks are a fundamental problem in C-like languages. Checkoway, et al. also describe a version of JOP that uses a similar "update-load-branch" scheme [15].

Sigreturn Oriented Programming

Signals are a powerful tool used in Unix-based operating systems to deliver messages to programs asynchronously. When a signal is delivered by the operating system to a running process, the operating system typically saves the running process's context on its stack and calls the registered signal handler for that signal (or ignores the signal if no handler has been registered). To restore the original program context, many operating systems like Linux configure signal handlers to transparently call a special `sigreturn` system call in place of executing a return instruction. This allows the original process context to be restored and resumes execution.

While `sigreturn` is designed to be called automatically, nothing prohibits user programs from invoking it just like any other system call, and there typically is no kernel verification that a `sigreturn` call is from a signal handler that is returning. Thus, as Bosman and Bos show in [11], it is possible for an attacker to create a fake process context on the stack and manipulate a program to call `sigreturn`. In

fact, `sigreturn` Oriented Programming (SROP) is in some ways easier to perform than ROP, as it only requires the location of the `sigreturn` gadget which makes the system call. This location, together with control over the instruction pointer, the stack pointer, and knowledge of the address of exploit code, is enough to launch a successful SROP attack on a vulnerable program.

In many kernels surveyed by Bosman and Bos, including Linux prior to version 3.3 on 64-bit `x86`, the `sigreturn` gadget lives in an area of memory not affected by ASLR, thus making its address static for the entire operating system distribution. To make matters worse, some versions of Linux place code to handle some time-sensitive system calls like `time` in a location of memory called the `vsyscall` page. This page is mapped into a static address in the user program's memory at startup, providing attackers with predictable, easy to find code to use as gadgets. This includes a system call and return gadget which simplifies many attacks. SROP is of course possible even on operating systems without static addresses for `syscall` or `sigreturn` gadgets.

Bosman and Bos also describe how to use SROP to leave a backdoor presence on a remote system and how to circumvent code signing on mobile devices. Like ROP with sufficient gadgets, SROP is Turing-complete, and shows us that ROP-style attacks are not confined to user programs (even in practice). They are much more difficult to protect than thought before, as all processes on Unix-based systems with just minor memory violations can be vulnerable.

Just-In-Time Return Oriented Programming

The Just-In-Time ROP technique (JIT-ROP) addresses the rise of defenses that rely on fine-grained address randomization like certain ASLR variants [52]. While quite a complicated technique in practice, JIT-ROP relies on ASLR's fundamental weakness to memory disclosure vulnerabilities. In a program with a memory disclosure vulnerability that allows access to a single function pointer, JIT-ROP is able to discover code addresses by disassembling the page containing that function pointer at runtime. JIT-ROP then recursively searches the set of memory pages it knows about for more code addresses. This way, JIT-ROP can discover large amounts of code in programs

that have already been subject to address space randomization. Then, by applying exploit techniques like gadget discovery and Just-In-Time compilation of exploit code at runtime, JIT-ROP is able to launch traditional ROP payloads.

JIT-ROP leverages common software engineering paradigms to wreak havoc on programs that use strong randomization-based defenses. In particular, programs that emphasize high code coverage may facilitate attacker discovery of large amounts of code at runtime in the presence of memory disclosures. Calls to system libraries can potentially reduce security even further, as they may expose runtime addresses of useful functions for attackers like `LoadLibrary` and `GetProcAddress` on Windows.

In summary, JIT-ROP reveals limitations of randomization-based defenses like ASLR, especially on large or dense programs. While possible countermeasures like runtime code re-randomization can be employed, JIT-ROP shakes any remaining confidence that ROP attacks are impractical to deploy on heavily-fortified systems.

2.2.3 Counterfeit Object-Oriented Programming

Counterfeit Object-Oriented Programming (COOP) is a code reuse attack that leverages semantics of the C++ programming language to achieve the expressiveness of ROP [47]. In C++, classes may declare that one or more of their functions may be overridden by functions in inheriting classes by designating them `virtual`. This allows code to treat objects as instances of their least-derived classes, and not worry exactly which function will be invoked at runtime (allowing dynamic dispatch). At runtime, the correct function to invoke is found through the use of a virtual function table (vtable) that is populated with the address of the function corresponding to the object's actual runtime type (i.e. its most-derived type). Each object that contains a virtual function or derives from a class with a virtual function contains a pointer to a such a vtable (vpointer). Virtual functions are a popular software engineering tool as they allow a degree of modularity between units of code.

COOP uses buffer overflows to insert "counterfeit" objects with attacker-controlled vpointers and data fields. Then, through the use of pre-identified snippets of virtual functions called "vfgadgets", existing vtables, and overlapping counterfeit objects, one

can initiate a code reuse attack using only the code from existing object's functions. COOP relies on the existence of specific but common vfgadgets, like a "main loop" gadget which iterates through a container of pointers to objects and invokes a virtual function on each of them. Note however that no code or vtables are injected into the vulnerable program. Instead, the counterfeit objects' vpointers are crafted to point inside existing vtables, allowing attackers to execute any existing virtual function.

Constructing a COOP attack requires significant analyses of the vulnerable program to identify useful vfgadgets, determine viable control flow between them, and arrange them (possibly overlapping) in memory to be inserted in the vulnerable buffer. While the COOP authors have designed a tool to help locate useful vfgadgets, significant human analysis is still required to generate a working attack. Attackers that can perform this work are rewarded with yet another exploit framework that is Turing-complete, but more automation of exploit creation is likely required before COOP attacks become commonplace.

The use of techniques like ASLR, DEP and stack canaries can make COOP attacks more difficult to perform, but do not conceptually prevent them. Defenses against ROP are also not necessarily applicable against COOP, as COOP does not alter existing return addresses or inject pointers to code. In short, COOP shows the true power of buffer overflows: any data that can alter control flow even in a passive way is a potential target for an attacker to modify through a buffer overflow.

2.3 Temporal Memory Vulnerabilities

The stack buffer overflow outlined at the start of Chapter 1 is an example of a spatial memory violation. Another important class of attacks relies on temporal memory violations like the use of heap-allocated memory after deallocation or the stack of a function that has already returned. In C and C++, dynamically allocated memory provides a particularly notable opportunity for memory violations as there are many ways to violate the contracts of `malloc` and `free` (or `new` and `delete` in C++). These include calling `free` on heap pointers multiple times, and the common "use-

after-free" error, where pointers to already deallocated heap memory remain in use. Because subsequent calls to `malloc` are allowed to reuse already freed memory, a clever attacker can use these pointers to read and corrupt memory at will.

A recent example of a use-after-free vulnerability is CVE-2014-1776 [14], which leverages a use-after-free vulnerability in recent versions of Microsoft's Internet Explorer along with supplied JavaScript and ActionScript code to achieve remote code execution. In this vulnerability, attackers fill the heap with many copies of their malicious code (a technique called "heap-spraying") and corrupt the saved length of a data structure to access all of memory. Then, they look for addresses of both the Windows `ZwProtectVirtualMemory` function, which attackers can use to make injected code executable, and a stack pivot gadget, which is used in conjunction with ROP code to set up the stack for injected code. In this attack the use-after-free vulnerability serves a similar purpose as a traditional buffer overflow, allowing an attacker to corrupt a data structure to access arbitrary memory.

Temporal memory violations are difficult to prevent, as C and C++ do not mandate garbage collection or schemes to invalidate pointers after calls to `free` or `delete`. ASLR, DEP, and canaries may make attacks based on temporal violations more difficult but do not attempt to detect or prevent the actual violation. Simple solutions include defensive programming, like setting pointers to `NULL` after freeing them, and the use of memory allocators that do not reuse memory, but these actions will not prevent all vulnerabilities and may have drawbacks like higher memory usage.

2.4 Other Vulnerabilities

One other interesting memory corruption vector is format string vulnerabilities. In C, functions of the `scanf` and `printf` families among others accept as arguments strings with format specifiers that allow interpretation of subsequent arguments in varying ways. Because C has a weak type system and these functions tend to accept a variable number of arguments, the number of arguments passed and their types are not checked at run-time. C also includes vulnerable format specifiers, including

specifiers that allow reading arbitrary stack memory and writing data to function arguments. These problems are exacerbated by the tendency of functions from the `printf` family to take a format string as the first parameter, which could be under the control of an attacker. Listing 1.1 in Chapter 1, for example, would also have a format string vulnerability if the vulnerable buffer were provided as the first argument to `printf` instead of the second. The interested reader is encouraged to take a look at Newsham's work on the subject [42].

C and C++ are languages ripe with opportunities for memory corruption, and this thesis does not attempt to enumerate all of them. Without memory safety, the possibility of new attack vectors involving memory corruption always exists. The increasing number of attacks on C-like languages imparts an increasing responsibility for programmers using those languages to not introduce bugs, which is not a desirable situation for anyone.

Chapter 3

Previous Work

Code reuse attacks are not a new phenomenon and affect a wide variety of programs. Unfortunately, there is currently no generally applicable low cost memory safety technique. However, there are quite a few proposed and implemented attempts at mitigating their impact on C programs. Because these solutions take very diverse approaches to mitigating attacks they provide different levels of safety, compatibility, and performance overheads. Naturally, programmers may find some solutions more appropriate than others. For example, mission-critical programs may require the highest level of security possible while being more flexible to modifying existing code or performance penalties. Some existing code-bases may be too large to accept code modifications, and as such would favor compatibility with existing source code or binaries over stronger security guarantees. For some use cases, no currently available solution may be acceptable.

This chapter outlines well-known defenses to code reuse attacks. In general, defenses to code reuse attacks and memory corruption in general require users to accept a trade-off between security, performance and compatibility with existing code. Not every solution is designed to prevent all code reuse attacks: some are even specific to classes of attacks.

3.1 Memory Safety Based Defenses

3.1.1 HardBound and SoftBound

Full memory safety is the prevention of all spatial and temporal memory violations. Techniques that provide full memory safety prevent against all of the code reuse attacks described in Section 2.2 as well as the temporal memory violations described in Section 2.3. (Note that memory safety does not imply prevention from information disclosure.) Two examples of memory safety techniques are the hardware technique HardBound [23] and its software counterpart SoftBound [40], both of which rely on the concept of "fat pointers". Fat pointers are merely pointers that are associated with a base and bound. With accurate fat pointers, ensuring spatial memory safety reduces to adding checks that ensure a pointer's value is between its base and bound before it is ever dereferenced ("bounds" checks).

A naive implementation of fat pointers can impose a serious memory bottleneck, as dereferencing a pointer suddenly requires three memory accesses. Implementations can also cause compatibility problems: C programs are very sensitive to memory layout and simply storing three words of memory for a pointer in place of one would break programs that assume the old pointer size. Fortunately, HardBound and SoftBound also apply intelligent techniques to reduce the number of memory accesses required for pointers. For example, while HardBound reserves space in virtual memory for base and bound metadata, it also reserves space for a small tag that encodes whether a memory location is a pointer or not. Memory locations containing pointers store their size in tags if it is a small power of two (like small C `structs`). On memory accesses it may suffice to check this small tag (which can be cached separately from other memory) instead of loading both base and bound for all pointers. HardBound and SoftBound also both remove bounds checks when they can be proven unnecessary at compile time. SoftBound is very effective in practice, detecting all memory violations in the Wilander test suite [66].

HardBound and SoftBound are able to provide provable safety guarantees without breaking the common C programming model [40], including precise control over

memory. Nevertheless, memory safety comes with a significant performance cost: the maximum runtime overhead HardBound displays in the Olden benchmarks [46] is over 20% [23]. SoftBound fares much worse, with an average case runtime overhead of 67% and multiple benchmarks with over 150% overhead [40]. Programs with fewer pointers suffer lower runtime overhead, but even the average-case overhead of 67% can be a steep price for programs to pay for memory safety. HardBound's significantly smaller overhead is enticing, but its required changes to hardware prevent its widespread adoption.

SoftBound's optional Compiler-Enforced Temporal Safety (CETS) extension provides highly-compatible, provably-correct temporal memory safety [39]. CETS provides temporal safety through the use of a global 64-bit counter called a "key" that is incremented on each call to `malloc` and also stored in the returned pointer. On calls to `free`, a "lock" (whose address is also part of the pointer) is unset. Then, temporal safety can be enforced by checking if the lock and key match. Unfortunately, CETS protections also come at a high cost: on the SPEC2000 [55] benchmarks the worst case runtime overhead was over 150%. CETS' temporal safety also cannot be guaranteed without the rest of SoftBound, so this overhead cannot be reduced for programs only desiring temporal safety.

3.1.2 CHERI and PUMP

Systems that are designed to enforce more comprehensive security models can easily and totally prevent code reuse attacks. One example is Capability Hardware Enhanced RISC Instructions (CHERI), a capability system that aims not just for exploit mitigation, but application compartmentalization [17] [65]. Based on the 64-bit `MIPS` architecture, CHERI introduces new hardware, changes to operating system kernels, compilers, the C standard library and introduces to programmers an object-capability programming model that allows enforcement of sets of fine-grained access policies on arbitrary program memory. The new capability coprocessor works in tandem with an MMU to provide intra-process memory protection at byte granularity. CHERI's protections are specifically designed to accommodate existing C programs and even

common C idioms that are implementation-defined or undefined behavior according to the C standard [17].

Like HardBound, CHERI uses a form of tagged memory to implement its security guarantees. Its purpose, however, is not for identifying pointers or storing their bounds. Instead, CHERI uses a one-bit tag on all physical memory to identify which locations contain valid capabilities. Capabilities themselves are stored in a set of dedicated registers. Capabilities allow much finer control of memory, including permission bits for instruction fetching, loading and storing of data and capabilities, and access to hardware exception registers. CHERI capabilities also include base and bound to allow for efficient bounds checks, allowing them to take the role of HardBound's fat pointers. The use of a one-bit tag to mark capabilities in memory helps them become unforgeable to attackers. Therefore, CHERI as a whole does not even need to leverage defenses like DEP and ASLR to effectively prevent against the spatial memory attacks presented in Section 2.2. CHERI also includes protections against temporal memory violations.

In *Architectural Support for Software-defined Metadata Processing* [24], a system called the Programmable Unit for Metadata Processing (PUMP) is described. The PUMP is a RISC processor with tagged memory, registers, and caches. PUMP represents the opposite extreme of the use of tagged memory: to allow for unlimited metadata, tags in the PUMP are large enough to store memory addresses of arbitrarily large metadata structures. Effectively, arbitrarily large numbers of memory policies can be enforced. Adding pointer-sized tags to all of memory does introduce many new costs, but the PUMP is able to utilize a number of optimizations when policies exhibit spatial or temporal locality, and policies defined by software themselves can be cached in hardware. The PUMP authors also show proof-of-concept policies that enforce memory protections schemes like DEP, full memory safety, and control flow integrity, among others.

Realistically, CHERI and PUMP are not currently viable options for most programs. While they are both highly compatible with existing C and C++ code, their use would require programmers and users to switch to an entirely new research ar-

chitecture, operating system, and compiler, among other components. They also rely on RISC architectures as a base, which presents different challenges than the reigning CISC `x86` architecture. It is also difficult to ascertain exactly what runtime penalty the use of such a system may impose on existing programs, as CHERI and PUMP provide such a wide range of functionality. What is not difficult is identifying CHERI and PUMP as systems that can prevent much more than memory corruption attacks that merit future attention, both to system programmers and hardware designers.

3.2 Annotated Languages

While C appeals to programmers that do not want sources of uncontrollable overhead like bounds checks automatically added to their programs, it makes the lives of programmers who are willing to pay for memory safety more difficult. Despite this fact, programming instead in a language with memory guarantees like Java may be undesirable or even infeasible for some programs. Annotated dialects of C like Cyclone [30] and CCured [41] are targeted at exactly this audience, providing a mix of static and dynamic checking that allow programmer control over memory corruption.

The Cyclone language exposes three types of pointers to provide control over bounds checks. Fat pointers are nearly identical to fat pointers in memory safety systems, but allow code to make dynamic inquiries about their bounds. Thin pointers are pointers without any bounds information, designed to be used as references to single locations of memory like integers. Bounded pointers are thin pointers with statically defined sizes, allowing programmers that use them to aid compilers in optimizing out unnecessary bounds checks. Like C, Cyclone allows for conversions between pointer types at will, but also allows programmers to write functions whose pointer arguments are guaranteed at runtime to not be `NULL`. Cyclone also uses a a region-based memory management system with a conservative garbage collector for heap allocations.

CCured, unlike Cyclone, is specifically designed for retrofitting existing C programs. Like Cyclone, CCured adds new types of pointers to C. SAFE pointers are similar to thin pointers in Cyclone in that they are intended to be references to single

values or `NULL` but also do not require the use of pointer arithmetic or casts. SEQ pointers are fat pointers that allow pointer arithmetic but not casts. SAFE and SEQ pointers are statically typed in CCured but WILD fat pointers can point to any type, like traditional C pointers. CCured also contains a number of minor pointer types like FSEQ pointers, which allow pointer arithmetic only with positive constants (to save the overhead of having to save the pointer's base), and runtime type information pointers which serve a similar purpose to their equivalent in C++. The CCured authors also make a significant effort to prove that their language facilitates development of programs without spatial memory violations.

Cyclone and CCured provide programmers with the opportunity to write memory safe programs without having to forgo the entire C language. The Cyclone and CCured authors are also very upfront about the amount of changes involved in porting C programs to their languages: the Cyclone authors noted that about 5-15% of code required changes in a port of a popular set of benchmark programs, while the CCured team only modified about 1% of code while porting a set of benchmarks including the Olden and SPEC95 suites [46] [54]. Notably, the CCured authors successfully ported a number of popular programs to the language, from programs crucial to security like OpenSSL and OpenSSH to hardware-interfacing programs like Linux kernel drivers. The performance impact of Cyclone and CCured is harder to measure as it depends on how programs are ported from C, but Cyclone achieves an average of 60% overhead on the Great Programming Language Shootout benchmark suite while CCured achieves an average of 32% overhead on a subset of the Olden and SPEC95 benchmarks [41]. CCured also measured its estimated memory overhead, which ranged from 1% to 284%. These overheads may be acceptable for certain types of programs given the power that Cyclone and CCured give to developers. For others, they remain too high, especially when combined with the time required to port existing code from C.

3.3 Intermediate Defenses

In this section, we explore a wide variety of solutions that aim to provide intermediate levels of security for all programs. Many of these solutions are designed to "fail open" – that is, protect as much code as possible while allowing unprotected code to run normally. Each has inherent weaknesses that reduce security guarantees, but all are effective for a wide variety of programs.

3.3.1 Intel MPX

Intel Memory Protection Extensions (MPX) [59] allows for hardware implementation of fat pointers on `x86` architectures. While HardBound stores base and bound metadata in virtual memory and uses tags to identify pointer and non-pointer memory [23], MPX creates four 128-bit registers to store base and bound along with hardware support for bounds directory and table structures (that are analogs of page directory and table structures). MPX also includes new instructions to efficiently check lower and upper bounds against pointers, which issue hardware exceptions when pointers are found to be out-of-bounds. The first architecture with support for Intel MPX is expected to be released later this year, so the following is an analysis of the Intel MPX simulator's behavior.

The CHERI authors discuss Intel MPX in a recent paper, comparing the extensions with full memory safety systems like HardBound [17]. Importantly, they highlight the fact that while both MPX and HardBound rely on explicit hardware instructions to handle pointer bounds, Intel MPX is designed to favor compatibility with existing programs over security. Two design choices in particular reduce MPX's security guarantees. First, Intel MPX allows unchecked dereference of pointers that have been updated without corresponding updates to their bounds. In this case, HardBound would issue a hardware exception on dereference. Second, because MPX metadata like bounds directories and tables are stored in separate memory, code that modifies pointer values must update bounds data separately which can lead to race conditions that permit out of bounds pointer dereference. In multi-threaded programs

this can negate MPX protection all together.

Because no Intel processors equipped with MPX have been released, it is difficult to draw any concrete conclusions about its properties, or how it will even be used by programmers (although `GCC` support for MPX has already been implemented [3]). Nevertheless, it seems that MPX will provide programmers with the ability to perform bounds checks more efficiently than before, and choose the level of memory protection they desire. However, in its current form MPX does not seem capable of providing full spatial memory safety and does not address temporal memory safety.

3.3.2 Heuristic Based Schemes

Processor state during a ROP attack can look very different than during normal execution. Recent low-overhead solutions like kBouncer and ROPecker try to identify precisely what makes ROP code execution so different through the use of several custom heuristics like gadget length [44] [16]. In particular, kBouncer and ROPecker use a hardware feature of recent Intel `x86` architectures called the Last Branch Record (LBR) to inspect the previous 16 indirect branches taken on system calls. While these tools have proven effective at preventing real-world exploits on existing binaries with very low performance overheads, the ease with which the heuristics they rely upon can be defeated leads to the conclusion that they do not provide strong security in theory. This section presents an overview of these heuristics and viable attacks against them, as originally discussed in [13] and [48].

As previously stated, kBouncer depends on the LBR to examine up to 16 previous taken indirect branches on system calls. kBouncer aims to enforce two properties on executing code. First, all previous return instructions stored in the LBR must return to instructions directly after call instructions. This heuristic is based upon the observation that many ROP gadgets use return instructions to return to places that were not intended as return points. Attackers that wish to subvert this protection are then limited to using call-preceded gadgets. Second, kBouncer checks the last 8 indirect branches to ensure that at least one of them does not go to code that has a path to another indirect branch in under 20 instructions. kBouncer calls these paths

"gadget-like", reasoning that a series of 8 such indirect branches is likely to indicate an ongoing ROP attack.

Unfortunately, both of these heuristics can be broken with some effort. In [13], Carlini and Wagner outline two attacks to break kBouncer. The first involves a ROP attack that sets up everything necessary to execute a system call but first executes a set of gadgets to alter the LBR's history. Only two gadgets are necessary: a history flushing gadget which does little work but is call-preceded, and a termination gadget that also does no work but contains at least 20 instructions before an indirect jump along every execution path. After these two gadgets execute, a system call can be made without detection. The second outlined attack uses only call-preceded gadgets, passing kBouncer's checks easily. Carlini and Wagner also show that sufficient call-preceded gadgets to perform such attacks are very likely to be found in programs of at least 70 kilobytes in size.

ROPecker uses a slightly different design. While also inspecting the LBR at system calls, ROPecker uses page permissions to keep a very small working set of executable pages. Before a page not in that set is to be executed, ROPecker ensures that there are not 11 "gadget-like" sequences of instructions about to be executed. ROPecker defines such a sequence as requiring 6 or fewer instructions before any path leading to an indirect jump. Nevertheless, the two outlined attacks on kBouncer can be updated to work on ROPecker by invoking gadgets that flush history more often and invoking termination gadgets multiple times. In short, because gadgets that perform these operations are present in programs of even a moderately-small size, attackers with knowledge of the heuristic used by a particular defense can work around it.

Heuristic-based defenses are very useful for end-users, as they allow a moderate level of protection for large amounts of existing programs without recompilation or large overhead. Unfortunately, they do little to solve ROP in practice, as attackers with knowledge of popular heuristics may be able to program around them. Additionally, defenses like kBouncer and ROPecker also suffer from conceptual weaknesses. For example, Intel's LBR is a global data structure: it is not saved and restored on context switches, which may make ROP attacks easier. Additionally, determining

suspicious execution using heuristics can disrupt legitimate programs [48].

3.3.3 Bounds-Checking Schemes

As discussed in Section 3.1, schemes that attempt to insert bounds checks into existing programs can carry significant performance and memory overheads. Baggy Bounds attempts to reduce these overheads by replacing the common fat pointer representation with a compact one-byte representation of pointer sizes [5]. Objects allocated with Baggy Bounds are padded to the nearest power of two, which itself can be stored in small bounds table. Then, bounds checks can be reduced to shifting a pointer over by its stored size and a single comparison instruction instead of the arithmetic and multiple compares required with fat pointers. Because Baggy Bounds uses allocation sizes instead of isolating individual object sizes, it does not prevent all buffer overflows. In particular, it is unable to detect overflows within C `structs`.

Baggy Bounds is able to speed up bounds checks, achieving an average of 60% runtime overhead on the SPECint 2000 C benchmarks. In part, this is because Baggy Bounds adds bounds checks on pointer arithmetic operations but not pointer dereferences. There are some notable drawbacks to this approach: in particular, Baggy Bounds does not allow all out-of-bounds pointers to exist, which can break programs containing out-of-bounds pointers that are never dereferenced. The SPECint benchmarks for example required modifications to work with Baggy Bounds due to this very issue. The use of a more compact bounds representation also results in significant savings in space overhead, with an average of 15% overhead on the same set of SPEC benchmarks.

Ultimately, Baggy Bounds and similar schemes that optimize bounds checks provide significant levels of security without intolerable overhead for many well-behaved C programs. While its overhead is still significant and its changes to memory allocations possibly unworkable for heap allocation-heavy programs, Baggy Bounds represents a practical midpoint between involved solutions like HardBound and heuristic based solutions like kBouncer.

3.4 Control Flow Integrity and Related Defenses

Control Flow Integrity (CFI) is a technique that aims for correct program execution rather than memory safety or integrity [4]. CFI relies on a statically generated control flow graph (CFG) that encodes all valid control transfers between a program's basic blocks. This entails determining where functions are called and where they return to, and adding checks to ensure these constraints are followed at runtime. CFI usually assumes that DEP is in place, and only instruments indirect control flow transfers. In theory, CFI is very resilient against ROP attacks, as they often call code rarely or never used in typical program operation.

Unfortunately, CFI is a difficult defense to implement. This is in part because the targets of indirect control flow data like function pointers are often difficult and in some cases impossible to determine statically. Code that loads libraries dynamically through functions like `dlopen` in `libc` also contributes to this problem. Another difficult problem is the determination of what can be a very large CFG. Two early CFI implementations, CCFIR and bin-CFI, consolidate program CFG's by conflating all indirect control transfers into a small number of "labels", distinguishing only function calls and returns [68] [69]. CCFIR also uses a third label for returns from system call wrappers. Implementations like CCFIR and bin-CFI have over time become known as coarse-grained CFI.

The research community has contributed much work on the effectiveness of coarse-grained CFI [53] [29] [21]. In short, coarse-grained CFI implementations like CCFIR use too few labels to be able to generate a precise enough CFG to prevent many ROP attacks. Even with a precise CFG, CFI lacks context-sensitivity, and as such is unable to determine if a function is returning to its actual caller instead of a function that calls it elsewhere. Nevertheless, coarse-grained CFI can be implemented on existing binaries with or without debug symbols and the ROPGuard implementation has been included as part of Microsoft's Enhanced Mitigation Experience Toolkit (EMET) despite its vulnerabilities.

More recently, researchers have focused on ways of overcoming CFI's limitations.

Cryptographically Enforced Control Flow Integrity (CCFI) attempts to provide fine-grained CFI using message authentication codes (MACs) for all control flow elements [36]. In CCFI, MACs of function pointers encode their position in code, allowing for more precise runtime indirect control flow checks. CCFI depends on recent Intel instruction sets with primitives for AES encryption to compute MACs efficiently.

Opaque CFI (OCFI) aims to prevent against memory disclosure attacks that weaken coarse-grained CFI defenses [38]. OCFI introduces the notion of bounds to destination sets of indirect control flow transfers, allowing control transfers only to addresses between the lowest and highest addresses in those sets. OCFI uses information-hiding techniques to hide these bounds structures from attackers, and accelerates runtime bounds checks through the use of Intel MPX. Unfortunately, the assumption that information-hiding techniques can prevent data discovery even in the presence of memory disclosure vulnerabilities can be a bad one [26]. Additionally, OCFI suffers from Intel MPX's weaknesses, like the lack of atomic updates to pointers and MPX bounds structures.

New implementations of CFI and solutions that use similar ideas are common. As an example, Forward-Edge CFI aims to achieve low-overhead protection by performing no instrumentation of returns and depending on solutions like canaries to protect them [63]. The Data Structure Analysis (DSA) algorithm [35] provides a strong static analysis framework for fine-grained CFI implementations. The lack of consensus as to the "right" version of CFI to enforce makes it difficult to evaluate its conceptual trade-offs on security, performance, and program compatibility. Despite this uncertainty, it is clear that CFI-based defenses merit further consideration before hard conclusions can be made about their true value to existing and future programs.

3.5 Memory Safe Languages

A potential solution for programmers desiring memory safe code is to leave the C language altogether. The recent rise of memory-safe languages like Rust that are designed for systems-level programming provide the potential for programs that are

free from code reuse attacks without requiring secondary software or new hardware [37]. This option, of course, is not without its drawbacks, particularly the risk of vulnerabilities in a language's implementation of memory safety impacting programs that use it, and the overhead involved in programming in an entirely new language.

3.6 Code Pointer Integrity

Like the techniques discussed in Section 3.1, Code Pointer Integrity (CPI) aims to protect memory via an expanded representation of pointers that includes base and bound [34]. Unlike those techniques, CPI aims to provide lower runtime overhead by only protecting "sensitive pointers". CPI defines these pointers as function pointers, pointers to sensitive pointers, pointers to types that include sensitive pointers (like `structs`), and universal pointers like `char *` and `void *` which are only protected when they become sensitive. By enforcing this selective form of memory safety, CPI is able to provide good security guarantees and low overheads for many programs.

Using mainly static analyses to identify these pointers, CPI instruments programs at compile time to place and access them in an isolated region of memory called the "safe region". Then, whenever a sensitive pointer is dereferenced, it is first subject to a bounds check. CPI provides spatial memory safety of sensitive pointers if the safe region is effectively isolated from attackers. In theory, CPI can also offer temporal memory protection, as the safe region can be used to store general metadata. For many programs, excluding non-sensitive pointers reduces the overhead of memory safety techniques to the point where CPI may be tolerable.

Although CPI's more limited form of protection is still strong, and it is able to protect against all attacks not protected by DEP or ASLR in the RIPE test suite [66], it suffers from a number of issues that prevent it from widespread adoption. These vary from high overhead for C++ programs that use many vtable pointers to weaknesses in its information-hiding scheme, which can be exploited by patient attackers through side-channel attacks [26]. CPI nevertheless is a source of good ideas for code reuse defenses, and it is analyzed in Chapter 4.

Chapter 4

Analysis of Code Pointer Integrity

This chapter outlines and analyzes the design, security guarantees, and overheads of the Code Pointer Integrity (CPI) code reuse defense. CPI's protections are affordable and appropriate for many programs, but can be difficult to implement securely on modern architectures. Nevertheless, CPI champions ideas that can be strengthened in hardware schemes like Taxi, which we introduce in Chapters 6 and 7.

4.1 Design

As introduced in Section 3.6, CPI [34] is a technique that aims to defend against code reuse attacks by moving "sensitive" pointers into a separate, hidden region of memory called the "safe region". All accesses to sensitive pointers are instrumented at compile time to access the safe region. Like SoftBound [40], CPI saves base and bound information for each protected pointer and performs bounds checks before dereference, thereby enforcing spatial memory safety. In theory, the safe region can be used as a general pointer metadata store, allowing CPI to potentially provide temporal safety.

Unlike SoftBound and similar fat pointer schemes, CPI does not view all pointers as crucial in preventing code reuse attacks. Instead, it defines "sensitive" pointers as pointers to types that can directly influence control flow. At minimum, this must include return addresses on the stack, function pointers, signal handler contexts,

`setjmp` buffers, and pointers to aggregate types like `structs` that contain function pointers. CPI goes a few steps further and and includes in their definition "universal" pointers like `char *` and pointers to sensitive types. This recursive definition covers structures like jump tables and C++ virtual table pointers, and is described by the CPI authors as an "over-approximation" of "code" pointers.

Because code pointers make up only a fraction of pointers used in many programs, protections limited to them can operate with significantly lower overheads. CPI further lowers its overhead by protecting return addresses separately using a shadow stack [34]. Unfortunately, exclusively protecting code pointers allows for considerable attacker control over programs. In the worst case, this can permit ROP attacks.

4.2 Security Guarantees

4.2.1 Threat Model

CPI's threat model allows an attacker full control over program memory outside of the code segment, which is loaded into unwritable memory. This includes the ability to read and write arbitrary memory through vulnerabilities, although executing injected code is prevented by DEP and variants of ASLR may be present. While this is a fairly strong attacker model, CPI notably trusts the compiler and loader to instrument and load the program into memory correctly.

4.2.2 The Safe Region

CPI provides effective spatial memory safety of sensitive pointers only if its safe region is isolated from attackers. Because CPI's threat model assumes that attackers can read and write arbitrary program memory, this is a challenging task.

The CPI prototype implementation Levee is designed to protect both 32-bit and 64-bit `x86` variants. In its 32-bit `x86` implementation, Levee uses an unused segment register to hold the address of the safe region and excludes access to this region through other segment registers by altering their segment limits. Unfortunately, while

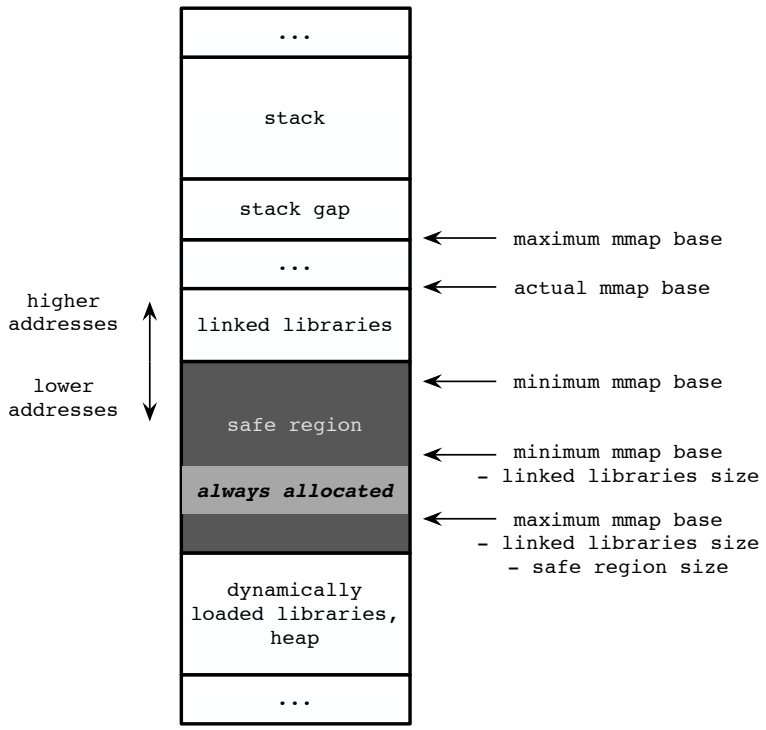


Figure 4-1: Memory layout of a process protected by CPI.

64-bit `x86` also provides segment registers, it no longer enforces their limits, preventing hardware isolation of the safe region. On this architecture Levee aims to protect discovery of the safe region's location by randomizing its base and ensuring that no pointers into the safe region exist outside of it. The CPI authors reason that this will force attackers to guess the location of the region, which in the presence of ASLR requires the impractical task of guessing addresses in a 48-bit region.

This claim is not entirely accurate. While it is true that discerning a single byte from the entire 48-bit `x86-64` address space is impractical, the safe region is much larger. In Levee, the safe region's size is chosen to be 2^{42} bytes and is allocated using the `mmap` system call and the `MAP_ANONYMOUS` flag. This means that its actual location is determined by `mmap`, and on systems that employ ASLR, the underlying operating system. `mmap` reliably aligns memory to page boundaries and only allocates below the stack gap.

Figure 4-1 depicts the possible locations of the safe region in memory. The amount of entropy in the page-aligned `mmap` base address depends on the underlying ASLR

implementation; as an example, the Linux kernel on 64-bit `x86` allows for 28 bits. Linked libraries are then loaded, followed by the safe region. Because the safe region is larger than the amount of ASLR entropy available, there will be a region of memory that will always contain part of it of size $2^{42} - (2^{28} * \textit{page size})$, where the page size is determined by the operating system. For the common case of 4KB pages this region is $2^{42} - 2^{40}$ bytes long, which spans several million pages. Therefore, attackers able to read and write arbitrary memory can easily corrupt code pointers stored in this region. By reading the values of those pointers, more patient attackers potentially could discover addresses of existing code like `libc`.

4.2.3 Locating `libc` and the Safe Region

The ultimate goal for attackers trying to circumvent CPI is to override control flow. In the presence of ASLR, attackers must first discover where useful code is located and then overwrite a code pointer. As Figure 4-1 illustrates, discovering the runtime location of a library like `libc` helps solves both of these problems, providing useful code and making it easy to locate the safe region itself. One simple but slow method to discover `libc`'s location is to use a memory read vulnerability to scan every address upwards from the always allocated part of the safe region for the first byte of `libc`.

Possible Optimizations

A faster method first described in [26] relies on the observation that memory at the minimum `mmap` base address is also always allocated, holding either the linked libraries or part of the safe region. By also using the fact that the `libc` allocation is page-aligned, the number of memory reads required to discover `libc` in the worst case can be reduced to the ASLR entropy available (in Linux, 2^{28}).

This is not the only optimization that [26] employs. We can rely on the layout of the safe region, which can hold up to 2^{40} pointers using its default linear "simpletable" layout. Because most programs have far fewer than 2^{40} code pointers, most pages in the safe region are entirely empty. Therefore, reading a zero page indicates that

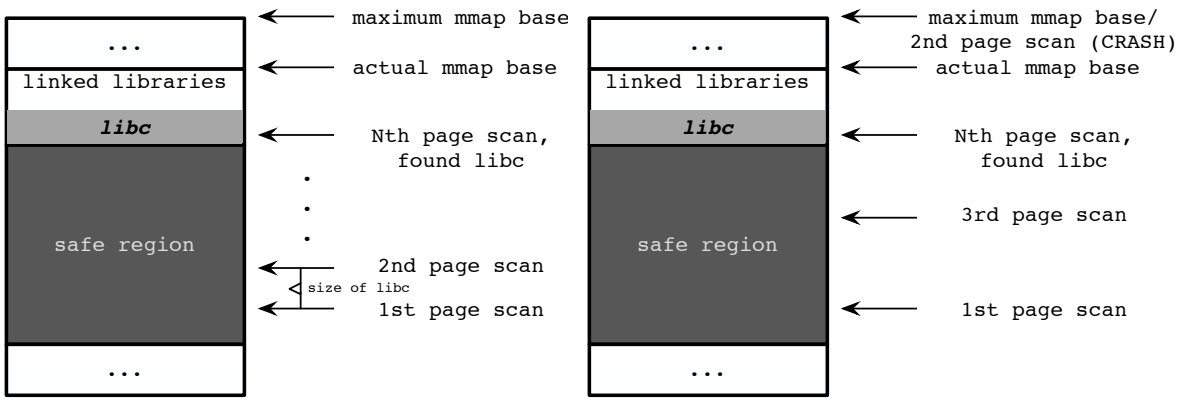
it most likely is not part of `libc`. If we believe this, in the worst case `libc` ends on the memory page directly above our scan. Then, instead of scanning the next page we can skip by the size of `libc` instead, reducing the number of scans we must perform in the worst case to $\frac{2^{28} * page\ size}{libc\ size}$, which is approximately 2^{19}. This strategy is illustrated in Figure 4-2a.

Tolerating Crashes

Many operating system kernels do not randomize the memory layout of processes started by system calls like `fork`, leaving programs that serve user requests with new processes especially vulnerable. In this case, discovery of the address space of a child process also provides an approximate layout of its parent. One side effect of this is allowing attackers attempting to guess program addresses the ability to guess incorrectly (likely crashing the child process) without disrupting the memory layout of the parent process.

This feature allows for further optimization of our strategy to discover the address of `libc`. Instead of starting our scan at the minimum `mmap` base address and skipping by the size of `libc`, we can perform a form of binary search where our upper bound address is out-of-bounds while our lower bound address is not. We initialize our search with our lower bound as the minimum `mmap` base address plus the size of linked libraries and our upper bound as the maximum `mmap` base address. When the difference between these bounds is approximately the size of `libc`, we have discovered it. This optimization reduces the number of scans required to $log_2\ 2^{19} = 19$, significantly speeding up our attack. Figure 4-2b illustrates this strategy.

One remaining question is how to identify `libc` during our memory scans. A simple method to do so when the version used by the vulnerable program is known is to create a table of individual `libc` byte values. When a non-zero memory page is discovered, scanning a few bytes in that page can then help fingerprint the exact `libc` page that was located. Then, discovering the address of a desired function or instruction is straightforward.



(a) Searching without any crashes. (b) Searching tolerating crashes.

Figure 4-2: `libc` searching strategies.

4.2.4 Constructing a Full Attack

In the presence of a memory disclosure vulnerability, we are now able to discover the base addresses of `libc` and the start of the safe region. With the ability to write to arbitrary memory, we can develop a full attack on a CPI protected program. Notably, the memory disclosure vulnerability does not have to be direct. [26] outlines a full attack on a CPI-protected version of the web server `nginx` that is able to manufacture a memory disclosure vulnerability through the use of a non-sensitive data pointer overwrite. This data pointer controls the number of iterations of a logging routine that is on the path of user requests, allowing attackers to determine byte values of arbitrary memory locations by analyzing network round trip time. This timing attack requires precise timing measurements and isolation of the baseline network delay.

Finding the Safe Region in Practice

The optimizations for discovering `libc` and the safe region in Section 4.2.3 depend on the assumption that any zero bytes discovered are within the very sparse safe region, and not `libc`. If we instead find zero bytes in `libc`, our searching strategies will skip over it and fail. One method of preventing this false-negative condition is to determine whether a page is entirely zero by reading page offsets that are never or almost never zero in `libc`. After experiments, we were able to find two bytes that are

never both zero in any page of `libc`, providing certainty in our search but doubling the number of accurate scans required.

Attacking the Safe Region

With the ability to write memory and the locations of `libc` and the safe region, we can both determine addresses of useful code (like system call wrappers) and write them to an existing code pointer's value, base, and bound entries in the safe region. Inevitably, we depend on already existing code to call the code pointer at some point to redirect control flow. To increase our odds of a successful and timely attack we can overwrite multiple pointer's values, which is easy to do because each pointer's metadata in the safe region is adjacent to each other and is organized identically. This does not carry the risk of crashing the victim program: the entire region has already been mapped into memory by `mmap`, allowing reads and writes even to unused virtual memory to succeed.

4.2.5 Static Analysis Weaknesses

CPI identifies pointers to protect via a compiler pass. There exist pathological cases where code pointers may be hard to identify in this pass, as C allows pointers to be cast in places static analysis cannot discover like dynamically loaded libraries. CPI will fail to identify and protect these pointers. SoftBound also has theoretical weaknesses in its static analyses but in its edge cases like casting arbitrary integers to pointers it requires programmers to manually provide bounds to such pointers before they can be dereferenced, thus erring on the side of caution. Because of C's weak type system, these issues represent fundamental weaknesses of static analysis based solutions and may be impossible to overcome without programmer annotations.

4.2.6 Defending the Safe Region

CPI's 64-bit `x86` implementation is vulnerable because it places the safe region in the same address space as other code, opening it up to discovery by memory disclosure

vulnerabilities. This problem is exacerbated by the safe region's large size, which allows attackers to scan memory to discover its base address without causing any crashes. There are several ways CPI can make discovering the safe region's base address more difficult.

The safe region could be placed at an address randomized independently of ASLR. This would make finding the safe region involve a great deal more guesswork. Unfortunately, `mmap` provides a limited amount of entropy in the address space available for fixed mappings, and because of ASLR, there is no guarantee that such a safe region would not collide with existing memory mappings.

A smaller safe region would be more difficult for attackers to discover, and could eliminate the always allocated region of memory highlighted in Figure 4-1. However, if the safe region were still allocated directly after the loaded libraries, our existing scanning techniques could still be applied in the reverse direction by an attacker with knowledge of any library address. Smaller safe regions also limit the amount of protection available to programs.

A small, externally randomized safe region that is non-contiguous with loaded libraries would evade all of our scanning strategies. Attackers would have to be willing to tolerate a potentially large number of unallocated memory dereferences to discover such a region. Nevertheless, there are viable ways to lower address entropy, like forcing the allocation of large data structures at runtime, and the use of large pages. This strategy is also not immediately viable, as common operating systems do not provide non-contiguous randomized memory allocators or allow user programs to construct this functionality in a reliable way.

Finally, the safe region could be partitioned into smaller, non-contiguous regions surrounded by pages with no permissions. Managing these regions would be more difficult, but making attackers have to perform multiple searches to identify the entire safe region could be worth the overhead. This strategy is pictured in Figure 4-3. For even further effect, these small regions themselves could be broken up into multiple pieces separated by garbage data at the cost of more management overhead. These strategies would also require the use of custom non-contiguous memory allocators.

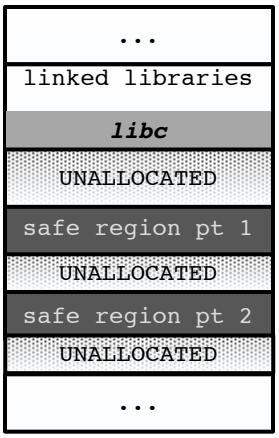


Figure 4-3: A possible non-contiguous safe region layout.

4.3 Overheads and Compatibility

Because CPI only protects sensitive pointers, it can reduce the performance overhead of full memory safety defenses like SoftBound to less than 10% for many programs. The CPI authors claim an average overhead of 8.4% on the C and C++ SPEC2006 benchmarks with information hiding [56], using an Intel Xeon machine running Ubuntu 12.04 with 512GB of RAM and a 2MB system page size.

While we did not have access to a similar machine to reproduce results, we were able to compile and run SPEC2006 on an Ubuntu 14.04 machine with 4GB of RAM and a 4KB system page size. All benchmarks were compiled using the Levee 0.2-instrumented `clang` 3.3 and the `std=gnu89` flag, which the `400.perlbench` benchmark requires to run. Table 4.1 presents our results using both the CPI default "simpletable" and hashtable safe region layouts, while Table 4.2 presents a comparison on select benchmarks between our results and those claimed by the CPI authors.

4.3.1 Experimental Methodology

The precise conditions that the claimed overheads in Table 4.2 were obtained with are unclear. The CPI team mentions that the simpletable layout with 2MB memory pages is the most performant in their tests, but it is not clear if their results use this setup. Additionally, no information on compiler flags passed to benchmarks

Benchmark	Language	Simpletable	Hashtable
400.perlbench	C	*NT*	*NT*
401.bzip2	C	1.42%	-0.35%
403.gcc	C	5.99%	*NT*
429.mcf	C	-6.55%	-3.47%
433.milc	C	-0.14%	12.9%
444.namd	C++	2.51%	3.05%
445.gobmk	C	2.53%	1.4%
447.dealII	C++	23.9%	24.1%
450.soplex	C++	2.28%	6.34%
453.povray	C++	*IR*	*IR*
456.hmmer	C	2.08%	1.04%
458.sjeng	C	5.2%	0.37%
462.libquantum	C	12.11%	11.01%
464.h264ref	C	16.02%	14.46%
470.lbm	C	5.98%	5.53%
471.omnetpp	C++	94.67%	131.1%
473.astar	C++	8.39%	0.63%
482.sphinx3	C	-3.59%	-1.27%
483.xalancbmk	C++	67.61%	*NT*
Average	C/C++	14.14%	13.79%
Average	C	3.73%	4.16%
Average	C++	33.23%	33.04%

Table 4.1: SPEC CPU2006 Benchmark Performance. *NT* denotes that a benchmark did not terminate after running for 8 hours, while *IR* denotes that a benchmark completed but its output was incorrect.

Benchmark	Language	Measured Overhead	Claimed Overhead
403.gcc	C	6.0%	16%
447.dealII	C++	23.9%	3.7%
453.povray	C++	*IR*	43%
464.h264ref	C	16.0%	5.8%
471.omnetpp	C++	94.7%	44.2%
483.xalancbmk	C++	67.6%	37%

Table 4.2: Comparison of measured and claimed CPI performance overhead using the simpletable safe region layout. Claimed overheads are estimated from Figure 3 in [34], except `447.dealII` and `464.h264ref`, which come from Table 3, and `471.omnetpp`, which comes from Table 1.

like optimizations and C/C++ standard levels are provided. The C standard we used (GNU 1989) was required to run `400.perlbench`, but its definition of function inlining can affect performance numbers for other benchmarks.

4.3.2 Reproducibility of Results

On the whole, the performance overheads described in the simpletable column of Table 4.1 are not radically different from the results claimed by the CPI authors: most benchmarks have similar magnitudes of overhead. Unfortunately, several benchmarks were nonfunctional in our tests. In particular, logical errors are introduced into the `453.povray` benchmark and several benchmarks like `400.perlbench` do not terminate when run for at least 8 hours. Implementation flaws in the available version of Levee may be to blame for this.

4.3.3 Analysis of Results

The CPI authors' claim of low overhead holds true for many benchmarks. Only five of the seventeen benchmarks that we observed to execute correctly had runtime overheads of over 10%. A few benchmarks even had improved performance under CPI: `429.mcf` ran about 6.5% faster. It is difficult to conclude why this occurred without knowledge of the benchmark's implementation, but it is likely due to the pointer memory locality that CPI creates: in programs that use sensitive pointers from different sources, CPI's safe region can effectively group them into a small region of memory, often residing on the same page. The use of larger memory pages may exacerbate this effect.

C++ benchmarks that use many objects with virtual functions are particularly susceptible to high overhead. This is because a sensitive pointer to a virtual function table exists at least once for each such object. Virtual functions are a core feature of object-oriented programming in C++, and their overhead limits CPI's applicability to user programs. The CPI team's answer to this overhead is a leaner version of CPI called Code Pointer Separation (CPS). Programs using CPS benefit from sensitive

pointers living in the safe region but those pointers are not subject to bounds checks. Additionally, CPS does not include pointers to sensitive pointers in its definition of sensitive pointers, excluding virtual function table pointers from protection. This improves the claimed performance overhead of C++ benchmarks to well below 20%. We do not evaluate the value of this security trade-off here, except to note that CPS is vulnerable to attacks like Counterfeit Object-Oriented Programming [47].

A small number of benchmarks performed significantly below expectations. Outliers include `471.omnetpp` with more than twice the runtime overhead claimed, and `447.dealII` with overhead fives times larger than expected. C++ benchmarks tended to have the highest overheads, with an average overhead of about 33%. The one C++ benchmark that we were unable to run successfully, `453.povray`, was reported by the CPI team to have 43% overhead, in line with our average.

Because it is uncertain exactly how the CPI team conducted their experiments, there are no definitive explanations for this behavior. However, one plausible scenario is that the increased memory usage of Levee-instrumented programs poses a problem for our test machine, which has only 4GB of RAM available. Paging would explain the particularly high overheads for `471.omnetpp` and `483.xalancbmk`, which had 36.6% and 27.1% of memory operations instrumented [34]. Testing CPI on a machine with a larger but realistic amount of memory would help determine if this is the case.

4.3.4 Source Code Compatibility

As described in Table 4.1, Levee 0.2 is not entirely compatible with the SPEC2006 benchmark suite. Because Levee is a prototype and the CPI team did not report these issues, this alone may not significantly limit CPI's generality. However, the size of the safe region can impact the level of protection available to programs and its overhead. To measure exactly how important this choice is, we ran SPEC2006 with the hash table safe region layout at three different sizes: the default of 2^{33} entries, 2^{26}, and 2^{20}. Table 4.3 presents benchmarks that failed at smaller sizes.

The Levee hash table implementation prints a "hash table full" message and aborts if it exceeds its linear probing limit while inserting a code pointer. While in principle

Failed at 2^{26}	**Failed at 2^{20}**
433.milc	433.milc
447.dealII	445.gobmk
450.soplex (*IR*)	447.dealII
471.omnetpp	450.soplex (*IR*)
473.astar	464.h264ref
	471.omnetpp
	473.astar

Table 4.3: SPEC2006 benchmarks that failed to complete by CPI hash table size. Unless otherwise noted, failures were due to CPI aborting with a "table full" error.

a poor hash function could lead to this error message, the safe region was quite full in practice, and the maximum number of concurrently present pointers in a safe region hash table in the SPEC benchmarks was observed to be around 2^{23} (occupying 2^{28} bytes). Thus, a hash table with 2^{20} entries is definitely too small for some benchmarks. It is more difficult to explain why some benchmarks using the hash table with 2^{26} entries failed, but implementation flaws are a likely cause. Implementation flaws most easily explain the incorrect output of the `450.soplex` benchmark under small hash table sizes, which does not occur at the default size.

Oddly, decreasing the hash table size resulted in lower overhead for some benchmarks. `401.bzip2`, for example, performed faster as its table size decreased, and under a hash table with 2^{20} entries was 8.6% faster than the default size and 7.3% faster than no CPI protection at all. Most benchmarks however only showed slight improvement with smaller hashtables and some even saw increased overhead, like `429.mcf`. Due to the failures in Table 4.3, these improvements must be understood with the knowledge that a fully correct implementation could behave differently.

In short, Levee suffers from compatibility issues that prevent full understanding of its performance and memory overheads. Because the number of pointers that are considered sensitive can be quite large, small safe regions can cause issues with real-world programs. Larger safe regions are more compatible, but are easier to discover using our information hiding attacks. While a very general technique, CPI still requires programs to accept a trade-off involving security, performance *and* compatibility. Further research and a correct implementation are required for lasting conclusions.

4.4 Lasting Ideas

Code Pointer Integrity relies on the observation that code pointers are the ultimate target of common code reuse attacks. By defending only these pointers, a large amount of safety can be obtained at much lower overheads than full memory safety defenses like SoftBound. While CPI's current canonical implementation has its flaws, this observation is a very valuable one and deserves to be explored further.

CPI highlights the utility of maintaining metadata on memory that is stored separately. In CPI, this metadata is used to provide spatial safety for code pointers. If the CPI safe region is thought as a general metadata store, it can be used to provide additional features like temporal safety. By isolating this metadata store effectively, strong security guarantees can be made. On 32-bit `x86` for example, the use of hardware segmentation to isolate the safe region effectively prevents the attack outlined in Section 4.2. Hardware proves to be an effective means to isolate memory, and metadata-based security solutions would be wise to use it when it is available.

On specialized hardware, CPI's security guarantees can be strengthened, and its performance overheads minimized. Nevertheless, CPI was designed to be a software solution, and its concept of a large, contiguous safe region is closely tied to software abstractions like virtual memory. We note that a software fault isolation version of CPI that may provide lower performance overhead is currently pending. However, when hardware is considered to be malleable new abstractions can be created that better serve CPI's security goals. One such promising abstraction is tagged memory, where all memory consists of contents and a metadata *tag*. Depending on their size, tags can encode various information. CHERI [65] uses a one bit tag that marks pointers, while the PUMP [24] has unlimited length tags that are defined by software.

In Chapters 6 and 7, we introduce the design and implementation of Taxi, a tagged architecture that implements fine-grained security policies that are also lightweight. Unlike CPI, Taxi is not limited to protecting code pointers; instead, it enforces policies on arbitrary memory. Like DEP, Taxi's hardware nature allows for security guarantees that are more resilient to attacks.

Chapter 5

Introduction to Tagged Architectures

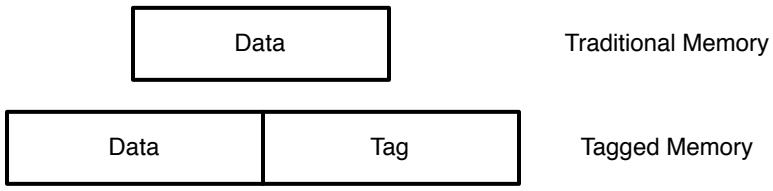


Figure 5-1: Traditional and tagged memory. The size of tags and the granularity of memory they are applied to varies significantly.

As introduced in Section 4.4, tagged architectures logically add metadata known as *tags* to all memory; this is depicted in Figure 5-1. Because the size, layout, and content of these tags can vary significantly, tagged architectures have been applied in systems with diverse goals and feature sets. Tagged memory provides a basis for the code reuse defense Taxi, which is described in Chapters 6 and 7; we introduce the concept here.

5.1 History

Tagged architectures are an old idea, having existed at least since 1957 [27]. Early tagged machines were used for various purposes. The Rice Computer R-1 from 1959 employed two tag bits that would trap the processor when set, designed to facilitate debugging. The IBM 7040 released in 1963 had a parity bit that could be set by

software that was commonly used to identify uninitialized memory. A more well known architecture was the Burroughs B5000 machine, released in 1960 [27]. The B5000 used a single tag bit that identified the start of memory regions subject to hardware policies defined elsewhere.

A common goal of early tagged architectures was hardware-level typing. One of the earliest analyses of such architectures was performed by Feustel in 1972 [27]. In his analysis, Feustel finds that machines like the Rice R-2 and the Burroughs B7500 encode the types of arithmetic data and vectors using tags. He also proposes a tagged architecture with a fine-grained type system encoding not just the size of arithmetic data but also its purpose. Some of the 32 pre-defined types in his system include matrices, stacks, queues, semaphores, and even files. Feustel chose types in part by looking at what contemporary programming languages like ALGOL, Fortran, and PL/1 required or used often. To account for what he could not predict, Feustel also allowed for software definition of types in his architecture.

Machines designed to run the LISP programming language also commonly implemented typing in hardware. Developed starting in 1973, the MIT LISP Machine Project formed around a 24-bit tagged architecture that assigned types to tags at runtime. This allowed for an instruction set featuring instructions that dispatched on the types of their operands, simplifying assembly significantly. These instructions handled error conditions by performing logical checks in parallel. The MIT LISP machines also implemented basic memory protections in hardware like checks on array bounds. Knight's well-known master's thesis describes a 32-bit variant of this machine known as CADR [33].

The family of Ivory processors released by Symbolics in the late 1980's were 40-bit tagged architectures with 6-bit tags [25]. They incorporated many of the hardware data types suggested by Feustel but also included complex types for lists, instruction types, generic functions, and many different types of pointers. Because pointers could be "forwarded" to other pointers, tags could also be indirect. Unlike previous machines that could address half-words, the Ivory family's smallest addressable quantity was a word, so its tags only existed on full words. Tags also assisted in hardware features

like garbage collection.

5.2 Modern Security Research

While some early tagged systems like the CADR had security features like hardware array bounds checking, memory safety was not a common goal. Recently, research involving tagged architectures has gone through a resurgence, and a good deal of it focuses on security applications. For example, [6] focuses on how tagged architectures can help secure operating systems, including enforcing the principle of least privilege to kernel code. The tag in this system is a full 32 bits, encoding the origin and intended memory space for data, along with control bits.

Section 3.1.2 describes the CHERI and PUMP systems. CHERI is a capability machine that uses a one-bit tag on all physical memory to identify valid capabilities [65]. PUMP represents the opposite extreme, allowing for potentially unlimited length tags that can define powerful memory policies in software [24].

In [67], Zeldovich et al. use tagged memory to enforce application-level security policies on data. Not content to trust the operating system to manage tags, their Loki system introduces a security monitor to do so. Together with an operating system that can track information flow, this monitor enforces fine-grained discretionary and mandatory access control policies on memory. Loki allows for 32-bit tags on each 32-bit word of physical memory, providing for a large number of policies that avoid issues with virtual memory like aliasing. With the knowledge that tags can exhibit high spatial locality, Loki also allows for tags that describe entire memory pages, reducing memory overhead when they are not needed. Loki requires significant new infrastructure including the creation of a permission cache and a compatible operating system but provides key insights into how a system with large tags can provide security to existing programs without unpleasant performance and memory overheads.

The dynamic information flow tracking scheme [57] by Suh et al. also provides insight into how tagged memory can be used to detect memory corruption. By effectively tracking foreign input from functions like `fgets`, many code reuse attacks

like ROP can be detected. This scheme uses a one-bit tag on each byte of virtual memory, on most registers, and on disk that marks whether data is potentially malicious. Like in [67], tag granularity is managed per page, here allowing tags per page, quad-word, and byte. Similarly, separate tag caches and TLBs help improve memory overheads. Finally, Suh et al. also provide detailed rules on how to propagate tags through normal arithmetic, memory access, and pointer arithmetic instructions.

Recent research also describes different ways to actually implement hardware tags. In [51], Shioya et al. describe a tag table structure that manages tags similar to pages. Like [67] and [57], it allows tags that describe entire pages of memory but uniquely only maintains tag information for allocated virtual memory. The tag table has multiple levels, allowing tags to be stored at each level and compressing the representation for tags on large regions of memory that use the same tag. This scheme uniquely allows for tag representation to change dynamically, growing and shrinking as needed.

5.3 Modern Implementations

Unfortunately, the successes of this security research have not translated into a popular consumer tagged architecture. There are efforts in this direction: the lowRISC project [12] aims to provide parameterizable tagged memory on top of the open-source RISC-V architecture [8] and has a working proof of concept. In lowRISC, users can enforce their own security policies through tag load and store instructions.

Commercially available specialized architectures sometimes use tags for limited purposes. Like HardBound [23], the IBM system i architecture [18] uses a one-bit tag to identify pointers in physical memory. A one-bit tag is also used in the Intel i960 to identify memory owned by the kernel [19].

Non-tagged architectures sometimes have extra bits available for use by software. For example, the RISC-V architecture ignores the lowest significant bit for code addresses, as all instructions are aligned to at least two bytes. This bit is still preserved in memory and can be used by programmers for marking function pointers.

Tagged architectures have experienced a resurgence through their applications to security domains. They are an attractive mechanism for systems that require compatibility with existing code in part because of their previous lack of popularity: legacy programs that have no interaction with tags can often be run in new systems using a default or empty set of tags. Tagged architectures are effective for creating new program semantics without altering memory layouts expected by existing code and are the basis for the code reuse defense Taxi described in Chapters 6 and 7.

Chapter 6

Defeating Code Reuse Attacks with Taxi

In this chapter, we use the idea of tagged memory described in Chapter 5 to develop Taxi, a small set of hardware modifications that help protect against code reuse attacks. Taxi expands memory to include tags, defines rules on how they are propagated by instructions, and allows their modification by software. Unlike previous hardware schemes that use large tags to enforce complex security policies like Loki [67] and PUMP [24], Taxi uses small tags and aims to provide protection with as few changes to existing hardware and software as possible. While this approach is not as well suited to defining custom security policies in software, it allows for a simple hardware definition of policies that efficiently prevent memory corruption.

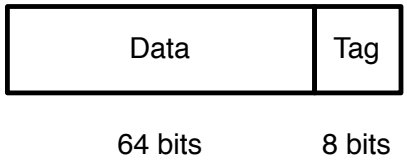


Figure 6-1: 8-bit tags are assigned to each 64 bits of memory in Taxi.

6.1 Use of Tagged Memory

As Taxi is a tagged memory scheme, registers and physical memory are logically divided into data and tags. As depicted in Figure 6-1, Taxi assigns an 8-bit tag for each 64 bits of physical memory. To preserve compatibility with programs that expect existing memory layouts, tags are located separately from data in virtual and physical memory.

A straightforward implementation of Taxi would use about 11% of physical memory for tags, but this can be reduced if the actual number of bits used by implemented policies is smaller. The policies we outline in this chapter use only a fraction of these bits and can be implemented in architectures where fewer are available.

All instructions that access or write data in registers or memory implicitly manipulate the tags associated with that data. In the worst case, this can require two accesses to main memory, doubling the memory overhead of those instructions. To avoid this, we employ a new caching strategy that stores tags with data in the first two level caches, which are widened to avoid reducing their capacity for regular data. We also create a dedicated tag cache that handles second level misses to further reduce the latency of tag requests. This memory hierarchy is illustrated in Figure 6-2. To evaluate how the size of this tag cache affects its hit rate, we have simulated tag cache behaviors at 11 sizes on the SPEC2006 benchmarks [56]. We present these results in Section 7.4.

6.2 Processor Execution

Because the results of processor instructions can propagate to both registers and memory, all instruction outputs must be paired with a tag. To enable efficient computation of such tags, a new tag processing unit (TPU) is created. Like a typical ALU, the TPU calculates its outputs by applying an arithmetic or logical function determined by the current instruction opcode to its inputs. These inputs are available at the same time as ALU inputs, allowing parallel computation of ALU and TPU

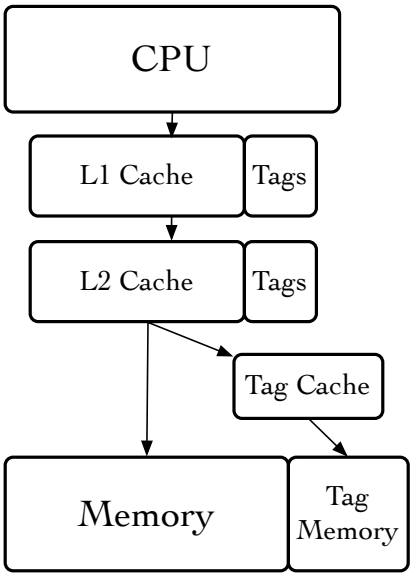


Figure 6-2: This system diagram outlines how tagged memory is laid out in Taxi. Tags are integrated into the L1 and L2 caches. A dedicated tag L3 cache also reduces the latency of tag requests and can be smaller than traditional caches due to a larger cache line size to tag size ratio.

outputs. With a fast enough TPU, the additional overhead to calculate result tags should be minimal for all instructions, as tag outputs will be available at the same time as ALU outputs. The role of the TPU in instruction execution is presented in Figure 6-3.

Unlike the well-defined arithmetic functions that an ALU performs, the outputs the TPU should generate are not obvious. The policies we will explore commonly use tags to mark data that can be used for control flow, assigning unprotected data a zero tag value. Because this means that a zero tag value is less privileged, we choose to implement a set of rules that allow the propagation of the least-privileged tag for some operations. These rules are described in Table 6.1. In particular, the exclusive-or operation is applied to the tags of inputs to addition and subtraction instructions to allow for policies that restrict other operations on pointers. Because other ALU operations are less commonly applied to pointers, simply taking the union of the tags of inputs can suffice. Most other instructions like memory loads and stores behave as expected, preserving tags when necessary.

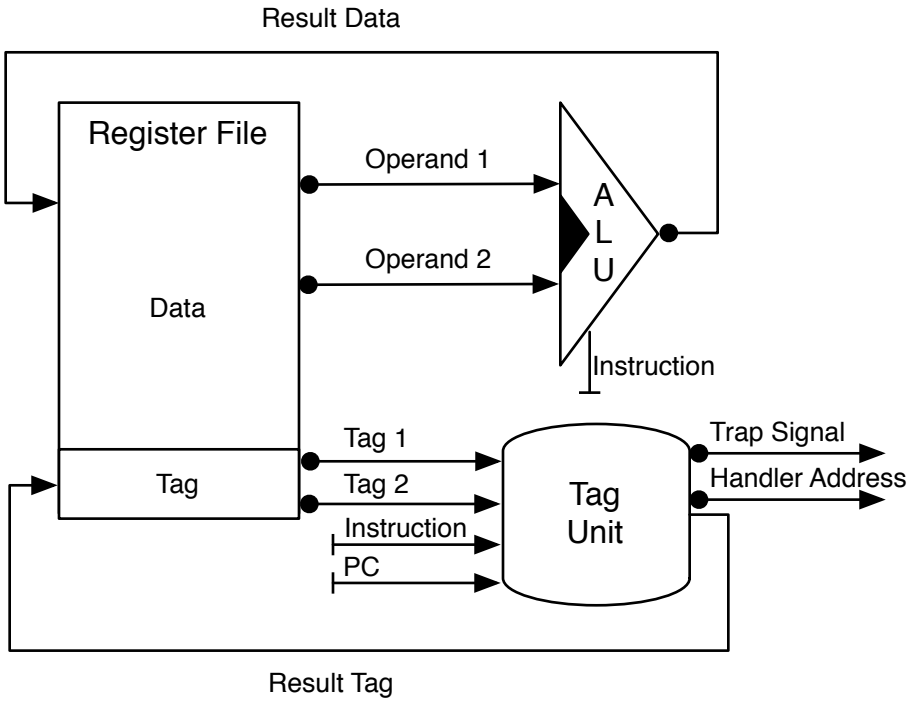


Figure 6-3: This pipeline diagram outlines how tags affect processor execution in Taxi. The dedicated tag unit allows parallel ALU and tag computation.

Operation	**Example**	**Tag Propagation**
Addition/Subtraction	`ADD RD, RS1, RS2`	`RD.tag = RS1.tag` $\oplus$ `RS2.tag`
Other ALU operations	`SRA RD, RS1, RS2`	`RD.tag = RS1.tag | RS2.tag`
	`XORI RD, RS1, Imm`	`RD.tag = RS1.tag`
Loads	`LW RD, RS1, Imm`	`RD.tag = Mem[RS1 + Imm].tag`
Stores	`SW RS1, RS2, Imm`	`Mem[RS1 + Imm].tag = RS2.tag`
Jump and Link (Call)	`JAL RD, Imm`	`RD.tag = TAG_PC`
Explicit Tag Set	`SETTAG RD, Imm`	`RD.tag = Imm`
Register Clear	`ADDI RD, R0, 0`	`RD.tag = 0`

Table 6.1: Taxi's tag propagation rules. Examples are written in RISC-V assembly. Registers and memory are treated like objects with `data` and `tag` attributes. `Mem[]` represents the value stored in the provided address. `R0` is a constant 0 register, and `Imm` represents an immediate value. `TAG_PC` is a special value for stored PC values.

The TPU is also responsible for enforcement of tag policies and does so by generating a trap when it receives unexpected input. Because Taxi's security policies are defined in hardware, the TPU is provided with all information required to determine policy violations including the current instruction, the processor's current privilege level, and the expected tag value for its inputs. By generating a hardware trap, the question of how exactly to deal with policy violations can be left to software, which benefits from more information about the current state of the program and more decision power. In Unix-based operating systems, this can be implemented using a signal with a default handler that exits the program. With this mechanism, software can log the exact instruction where the violation occurred, kill the offending process, or raise a system-wide alarm.

6.3 Protection Model

In Taxi, tags are created and maintained by instructions at runtime according to a list of hardware defined policies. This allows programs to gain protection without being rewritten or recompiled with a tag-aware compiler. When such a compiler *is* present, additional tools like instructions that manually set tags can allow for policies with statically defined elements. In principle, Taxi can also support policies that require an initial set of program tags to be loaded into memory with a compiled binary.

Taxi places trust in the operating system to maintain and occasionally manipulate the tags of running programs. Because of this trust and the fact that kernels are less suited to our security policies, Taxi only enforces policies for programs while running in user-mode. This frees policies from concern over exceptional control flow patterns like context switches and interrupts. Tags are still created and propagated during kernel execution, as they may propagate to user-mode code.

The key goal of Taxi is to provide metadata that cannot be interfered with by attackers. Because plausible attacks exist on metadata defined in software like Section 4.2's attack on Code Pointer Integrity's safe region, Taxi provides its metadata in hardware where more solid guarantees can be made. In particular, the rules on tag

propagation in Table 6.1 cannot be altered without further modifications to hardware, making the job of attackers wishing to write arbitrary tag values difficult. While an explicit instruction to set tag values (`settag`) exists, it only applies to registers using an immediate encoded into its opcode and is for most programs extremely rare, only appearing in policy edge cases that are insulated through other means. Taxi's tag propagation rules are also atomic, preventing attacks that exploit inconsistent metadata states which are a problem for defenses like Intel MPX [59]. When used with defenses like DEP, Taxi effectively defends against a large class of code reuse and code injection attacks.

6.4 Explored Policies

Here we describe policies whose security, performance, and compatibility properties we have explored in depth. We outline their implementations in Chapter 7.

6.4.1 Call/Return Discipline Protection

The function call and return discipline is one of the most fundamental abstractions in computer science and is provided for by many computer architectures in the form of explicit `call` and `return` instructions. While RISC architectures like RISC-V [8] do not provide these instructions explicitly, they provide a generalized jump-and-link instruction that allows for the same functionality. In either mechanism, function calls save the address of the following instruction for later use during function return.

Unfortunately, `return` instructions and equivalents typically do not perform any validation on their arguments, allowing for execution to be redirected to any valid location. This problem is more egregious on architectures that store return addresses on the program stack like `x86` because the location of return addresses is often assumed to be static. Even when protections like stack canaries are in use, return addresses are high value targets for memory corruption vulnerabilities.

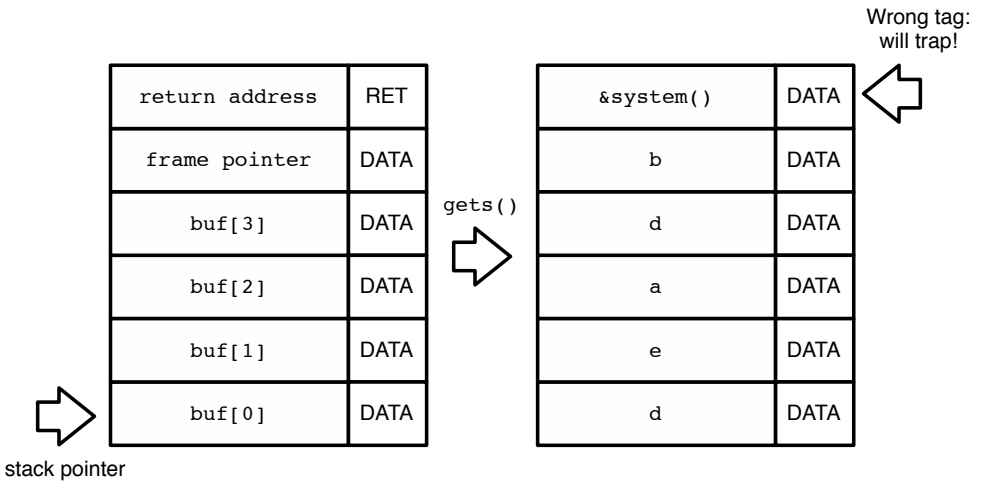


Figure 6-4: Diagram of the stack layout of a vulnerable C program under Taxi. When the `gets` function overflows the stack buffer `buf`, the corrupted return address does not have the return address tag, causing the machine to trap when it tries to return.

Security Guarantees

Taxi makes checks on return addresses easy to perform by marking each with a unique tag (`TAG_PC` in Table 6.1). If the TPU does not see this tag on a `return` instruction, it will generate a hardware trap. Because all valid return addresses are created at runtime only by `call` instructions, we can mark these addresses by instrumenting `call` instructions. Attackers that can neither generate these tags nor inject code are then restricted to only using return addresses that exist in the program from expected `call` instructions and cannot redirect control flow through return addresses. This is similar to the protection offered by a shadow call stack [4], but does not prevent using return addresses out of order.

Note that nothing in this policy prevents or detects buffer overflows: buffer overflows that overwrite data other than return addresses will succeed unless other defenses are present. This policy only prevents the actual control flow transfer required by code reuse attacks with return addresses.

As an example, a C program that includes a vulnerable four-character buffer on its stack is protected by the call and return discipline policy in a manner depicted in Figure 6-4. Although the return address is successfully overwritten here, attackers

are unable to write the return address tag required for a code reuse attack using it to succeed, as only `call` instructions are able to do so. This attack will be detected when the vulnerable function attempts to return, and a hardware trap that raises a catchable signal will occur.

A careful reader will note that by their nature, `settag` instructions have the power to write return address tags. For these instructions to be of any real use to an attacker, they must be in a vulnerable function after a buffer overflow at *compile time* and reference the correct register and immediate. This scenario is extremely unlikely, as return address tags are created and maintained without any compiler intervention, and `settag` instructions are only used in our policies in a small number of exceptional cases outlined in this chapter.

Program Compatibility

Because languages like C and C++ are extremely dependent on a correct implementation of the call and return discipline, most programs in these languages should have little trouble with its enforcement in hardware. One exception is kernels, which must be able to switch program contexts at will to implement features like preemptive multitasking. User-level programs also occasionally use `return` instructions outside of the call and return discipline; examples include signal handlers and C++ exceptions.

In C and C++, designated signal handlers are called immediately when a signal is received, pausing normal execution. Because this can occur at any point in code, the values of registers including the program counter must first be saved before running a signal handler. When signal handlers complete, the original values of registers must be restored. Typically, this is handled by the `sigreturn` system call that signal handlers call automatically. The mechanism for making this system call on Linux is kernel code that changes the return address of the signal handler at runtime. This procedure involves a `return` instruction returning to an address not generated by a matching `call` and must be trusted at runtime with the `settag` instruction for signal handling to function.

C++ exceptions present a situation similar to signal handlers. When a program

throws an exception, normal control flow ceases and an exception handler is run. Exceptions are usually implemented on modern systems with a two-phase procedure that determines the address of a compatible exception handler and then launches that handler by creating a return address and `return`ing into it. For an exception handler to function in Taxi, this return address must be trusted at runtime. Exception handling code is typically provided as part of a compiler-supplied runtime library, which Taxi must trust to not maliciously redirect control flow. This does not increase Taxi's trusted computing base because versions of these libraries provided by compilers, like `libgcc`, are already trusted by programs utilizing C++ exceptions.

Performance Impacts

While it is difficult to evaluate the minimum performance overhead of the call and return discipline policy, tagging return addresses requires almost no changes to existing source code and hardware instructions. The instructions that do require modifications, `call` and `return`, are augmented only to create and check for return address tags. While even slight overhead on these instructions could slow down programs with a large number of function calls or recursion, the TPU allows for efficient tag determination and checking, alleviating the most practical sources of overhead. This is especially true on RISC architectures, where jump-and-link instructions that implement function calls and returns typically only operate on registers, avoiding the need for costly memory accesses.

As this call and return policy is very simple, it seems likely that the dominant source of overhead for programs utilizing it will be the overhead required to support tagged memory. There are more than a few simple optimizations to Taxi that can improve its performance with return addresses. Two such optimizations are the introduction of a tag predictor and a targeted cache replacement scheme.

In many programs, return addresses only make up a small fraction of stack memory. To speed up the TPU, a dumb but fast tag predictor that only takes into account instruction opcodes could be introduced, allowing the TPU to provide the correct tag value when ready. A simple prediction scheme would guess the return address tag

for `call` and `return` and an empty tag for all other instructions. Because code reuse attacks are exceptional conditions, a tag prediction "miss" would be fairly infrequent. The cost of missing would also be lower than in branch predictors, as the correct tag could be filled in when needed for most instructions. Incorrect guesses on `return` instructions would be more troublesome, as they could result in unwanted control flow transfers. Nevertheless, this would be detected when correct tag values become available.

A cache replacement scheme for the tag cache that is aware of return address tags can decrease their overhead by favoring the removal of zero tags. In this scheme, the tags of return addresses can persist longer in fast memory, decreasing the likelihood that a tag lookup delays execution of a `return` instruction. This is especially effective when combined with the simple tag predictor, as it allows control flow attacks to be detected faster without penalizing memory accesses of data with zero tags. Future work would be required to evaluate the impacts of such a scheme on memory-intensive programs.

6.4.2 Linearity of Return Addresses

The call and return discipline policy is a lightweight, effective, and highly compatible way to use tagged memory to prevent code reuse attacks from corrupting return addresses. By marking return addresses with unique, unforgeable tags, `return` instructions are prevented from operating with arbitrary addresses. However, this policy does not enforce any constraints on when return addresses are used, allowing attackers to "replay" existing return addresses that are not removed from memory after use. The CFI defense suffers from a similar conceptual problem [4]. Copies of return addresses created by a compiler or programmers could also be used maliciously, as they also possess the return address tag.

Our approach to preventing malicious replay of return addresses is to minimize the use of the return address tag. We do this by ensuring that only one copy of each return address is usable at all times, a property we call linearity. When a return address is copied to a register or memory location, we *move* its tag instead. The

Operation	Example	Tag Propagation
Addition/Subtraction	`ADD RD, RS1, RS2`	`RD.tag = MRET(RS1.tag` $\oplus$ `RS2.tag)`
Other ALU operations	`SRA RD, RS1, RS2`	`RD.tag = MRET(RS1.tag | RS2.tag)`
	`XORI RD, RS1, Imm`	`RD.tag = MRET(RS1.tag)`

Table 6.2: Altered tag propagation rules for the return address linearity policy. `MRET(x)` denotes the result of masking out `TAG_PC` from `x`.

old return address itself persists in memory but is unusable by `return` instructions due to the missing tag. When `return` instructions operate on a register like in many RISC architectures, this prevents return addresses from being used more than once. These protections can be implemented in hardware by allowing the TPU to write to the tags of its input registers. Linearity of return addresses can also be implemented in tag-aware compilers by inserting `settag` instructions directly after all copies of return addresses, at the cost of more complicated static analyses and atomicity.

Tracking the movement of return addresses is straightforward for instructions that copy memory but is more difficult for functional instructions like ALU operations. To avoid having to instrument every such instruction to check for return addresses, we instead instruct the TPU to strip all return address tags involved in these instructions. This can be implemented efficiently by modifying the first two tag propagation rules in Table 6.1 to first mask out return address tags from inputs. The revised rules are presented in Table 6.2.

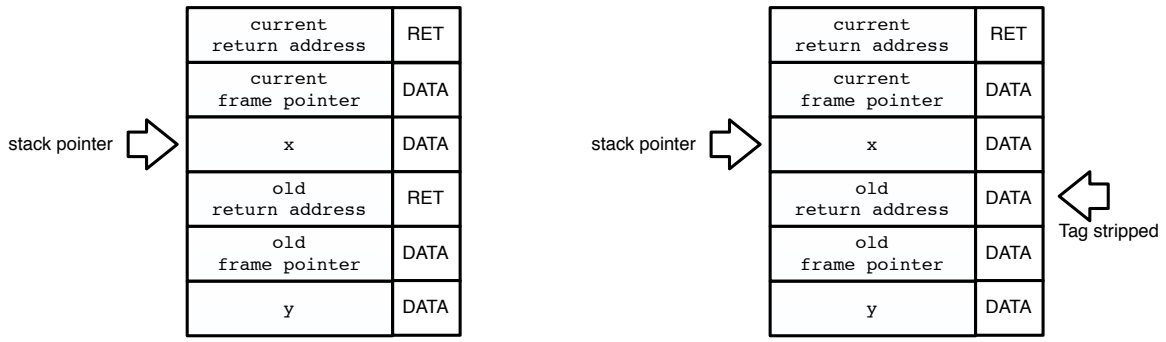
Together, these modifications minimize the possibility of return addresses being used for anything other than pure storage. These restrictions do not restrict any language features in C and C++, as those languages do not provide explicit mechanisms to modify return addresses anyway. By enforcing this policy in hardware, the possibility of our call and return discipline policy elevating the value of replay attacks on return addresses is eliminated.

Security Guarantees

The call and return discipline policy effectively limits the set of addresses that can be used with `return` instructions to the set created by `call` instructions (and copies thereof). The return address linearity policy builds on this restriction, further ensuring that only one copy of all return addresses created by `call` instructions can be used by a `return` instruction at a time. On architectures where `return` instructions operate on registers, this means that return addresses not saved after a `return` can only be used once – providing a much stronger guarantee for the common case. This guarantee can also be accomplished on architectures where `return` instructions use stack addresses implicitly like `x86` by manually masking the return address tag away. In either case, enforcing linearity of return addresses prevents programs from committing the temporal error of reusing old return addresses.

The primary motivation for this policy is the observation that in C and C++, no stack memory is cleared after function returns. This allows stack contents from functions that have returned to be used by their callers, despite the C standard not providing this feature. This memory is overwritten when the existing stack grows to use it, often on the next function call. Nevertheless, when the contents of stack memory are sensitive this can still pose a significant security risk. An ideal solution from a security standpoint would be simply zeroing a function's stack as it returns. When stacks are large or many functions are used in a program, this can require a level of overhead prohibitive for many programs. The return address linearity policy is a compromise between security and performance, protecting used return addresses with constant time operations. An example of a stack layout protected by the policy is provided in Figure 6-5.

To make enforcement of the linearity property easier to implement, we also prevent the use of return addresses that have been manipulated by arithmetic and logical functions as arguments to `return` instructions. This does limit the use of return addresses more than strictly necessary, but the corresponding loss of functionality is anything but significant. C and C++ do not provide portable ways to even acquire



(a) A stack frame layout without linearity. (b) The same stack frame with linearity.

Figure 6-5: The impact of the return address linearity policy on a stack frame that grows downward. Below the stack pointer lies the stack frame of a function with local variable `y` that has already returned. Without stripping the tag on that function's return address, it remains a target for a capable attacker.

the return address of a function (although programs compiled with recent versions of GCC can use the `__builtin_return_address` function), and no mechanism to alter them is provided in the language short of reaching into memory manually, which may not be possible on some architectures. In summary, return addresses are simply not designed to be modified between definition and use for most programs.

Program Compatibility

The linearity policy has no effect on programs where return addresses are not modified or moved between definition and use; we have observed this to be the common case. Occasionally, return addresses are copied and replaced before their use, especially on architectures that store return addresses in registers that are also used for storage of temporary variables. In these cases, the linearity policy only introduces problems when multiple copies of return addresses are created and the copy used with a `return` instruction is not the most recently created. As further described in Chapter 7, this situation was not observed in a wide corpus of programs.

As previously mentioned, the lack of a portable mechanism to intentionally modify return addresses outside of function calls in C and C++ means that programs that use modified return addresses as arguments to `return` instructions are rare. We were able to find only one example of such a program; it is a pathological case that is

nullified by compiler optimizations and is discussed in Chapter 7.

Performance Impacts

The return address linearity policy generates no additional tags and decreases the amount of memory with nonzero tags. The modifications to the tag propagation rules required to do so are also minor and should not create a performance bottleneck. Decreased tag usage in turn can mildly improve the performance benefits of optimizations like the simple tag predictor. In this respect, the linearity policy provides an opportunity for increased optimization of tagged memory.

Notably, implementing return address linearity without the aid of compilers requires the TPU to be able to write to its input tags when return addresses are copied. The ease with which this can be implemented depends heavily on the architecture that Taxi is applied to. When multiple register file read and write ports are available, instructions that read and modify tag values in one cycle can be used to strip return address tags. Architectures like RISC-V that copy memory through registers also help by mitigating the need for instructions that read and write tags in memory.

When implemented with a tag-aware compiler that can intelligently insert tag adjustment instructions when copies of return addresses are created, linearity can be enforced without new instructions or the creation of a new pipeline stage. Such implementations only impose overhead when return addresses are actually copied, which is uncommon for most programs. However, software implementations may not benefit from the atomicity of tag assignment that hardware can provide.

6.4.3 Restricting Partial Copies

While the return address linearity policy effectively prevents replay attacks, its protections can require a moderate amount of changes to hardware. To provide protection with fewer hardware modifications we introduce the partial copy policy, which removes return address tags from partial stores of return addresses. This prevents memory that only stores part of a return address from receiving the return address

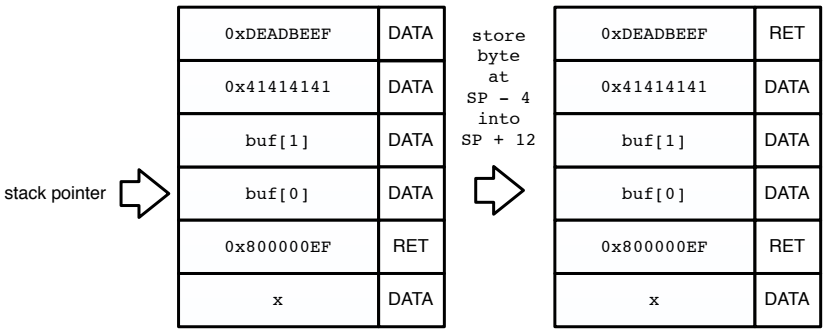


Figure 6-6: An attack prevented by the partial copy policy. Here the stack grows downward, and return address linearity is not enforced. The `0xDEADBEEF` address overflowed onto the stack can be legitimized by an attacker able to store the corresponding byte from an old return address.

tag. Because compilers typically move return addresses atomically, this restriction should have no effect on legitimate control flow.

An example of an attack that the partial copy policy prevents is illustrated in Figure 6-6. Because of tag propagation rules, attackers with the ability to copy individual tagged bytes can legitimize return addresses inserted during buffer overflows by using existing tagged return addresses. Here, an attacker can use the existing `0x800000EF` return address to legitimize the `0xDEADBEEF` address inserted during a buffer overflow.

This is a difficult attack to perform, as it requires knowledge of the values of previous return addresses and the ability to copy data *with* its corresponding tags. Nevertheless, because the cost of preventing it is minimal, the partial copy policy is a valuable addition to the Taxi arsenal. For more information on the specific security and performance guarantees of this policy, see [28].

6.4.4 Data Blacklisting

The partial copy policy operates on the observation that return addresses are usually loaded and stored all at once and thus are not touched by instructions that manipulate data at a smaller granularity. The data blacklist policy applies this observation to all pointers, asserting that in user-level programs sub-word level instructions primarily

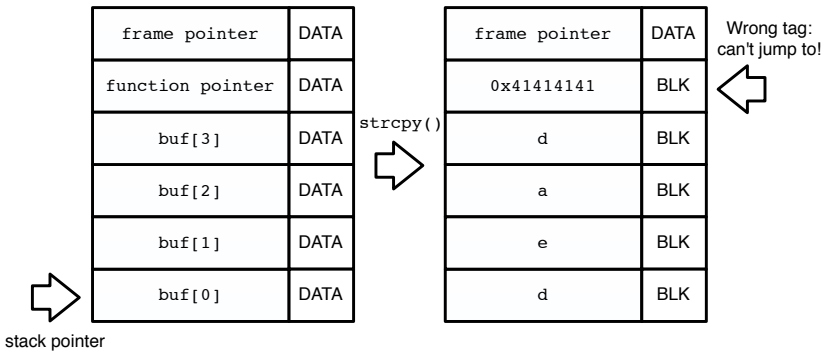


Figure 6-7: An attack prevented by the blacklist policy. Here, the stack grows downward. Because the function pointer adjacent to `buf` was written with `strcpy`, any attempt to use it as an address in hardware will trigger a trap.

act on data. It creates a new tag for non-pointer data that is subject to several restrictions. Data with this new tag cannot be jumped to or used as an argument to load or store instructions because such data can never represent pointers of any kind. Non-pointer data *can* be used for operations involving regular data like addition and subtraction, which are required for pointer arithmetic.

The policy of blacklisting data that has been manipulated at sub-word level provides valuable protections after buffer overflows occur. C functions that read strings into user-provided buffers byte-by-byte like `sprintf` and `strcpy` are commonly used to manipulate user input and thus are very vulnerable to such attacks. These functions however are rarely intended to facilitate the use of user input as program addresses; as such, it is appropriate to blacklist all data passed to them. By doing so, corruption of pointers adjacent to buffers that are manipulated with vulnerable string functions can easily be detected, as pictured in Figure 6-7. Here, the buffer overflow that modifies the adjacent function pointer will be detected when the function pointer is called.

The blacklist policy does not directly protect pointers from buffer overflows in all functions that deal with user input. Notable exceptions include `memmove` and `read`. When it *does* apply, it is able to provide protections similar to taint tracking and information flow-based solutions like [57]. Because sub-word memory accesses on data are common, the non-pointer data tag is used frequently, negating some

of the effectiveness of proposed optimizations to Taxi. Nevertheless, blacklisting is compatible with most of the programs we have explored. For information on the precise security, performance, and compatibility guarantees offered by blacklisting, see [28].

6.5 Other Policies

6.5.1 Universal Pointer Protection

One conceptually simple policy is to create a tag for all pointers. With the help of static analysis by a tag-aware compiler, all pointers are tagged at program start. Then, a set of tag propagation rules that allow pointer arithmetic with non-pointers but disallow most other operations are responsible for protecting tagged pointers. Memory accesses and function calls that act on addresses also require their arguments to be pointers. If correctly implemented, this policy can protect attackers from forging return addresses, function pointers, and pointers to function pointers like C++ virtual table pointers.

Unfortunately, determining the correct set of tag propagation rules to protect all pointers without breaking existing programs is a difficult task and may be impossible. Because the C type system allows for arbitrary casts to and from pointer types, statically tracking the movement of all pointers throughout a program to keep their tags accurate may be infeasible, especially for large programs. Even if tags are correctly tracked, there are many programs that use pointers in difficult-to-analyze ways. To support dynamically loaded code under this policy, functions like `dlopen` also require the ability to load tags into a running program. The level of protection that can be achieved through a universal pointer policy may be worth the difficulty of its implementation, but we leave such an implementation to future work.

6.5.2 Protecting Function Pointers

To avoid the difficulties of implementing a policy that protects all pointers, we can apply the same idea strictly to function pointers. Again, to do so, a tag-aware compiler will perform static analysis to identify and track the usage of function pointers throughout a program and ensure that function pointer tags are loaded at program initialization. The tag propagation rules necessary for function pointers are simpler than those required for the universal pointer policy as only basic function pointer arithmetic must be supported. When tags are properly propagated, indirect function calls can be restricted to addresses with function pointer tags. Despite the reduced benefits of this scheme, it can provide protections similar to CPI [34] more securely, atomically, and at lower overhead.

6.5.3 Protecting Jump Tables

Lightweight protection of data structures that hold function pointers is also possible with Taxi. Values of global jump tables with constant values like the Global Offset Table (GOT) can be marked with a unique tag at program initialization. By checking the appropriate tags when performing library function calls, attackers will be unable to forge GOT entries. C++ virtual function table pointers can be protected with a similar policy that also allows for tags to be copied; this allows C++ objects containing such pointers to be copied throughout memory. Implementation of either policy requires the use of a tag-aware compiler and a way of signaling addresses to the TPU whose tags must be checked during `call` instructions.

6.6 Design Exclusions

While the Taxi design outlined in this chapter provides an explicit `settag` instruction that gives software control over tag values, it notably does not include a corresponding instruction to load tags into memory. We found this instruction unnecessary to enforce any of our implemented policies, but it will be needed by the operating system to

serialize memory to disk during paging. Restricting the use of such an instruction in user-level code will help make attacker analysis of runtime tags difficult and is in line with Taxi's goal of defending code reuse attacks by limiting code that can be reused.

Kernels that serialize tags for paging to disk also need a mechanism to move tags efficiently. They would benefit from general purpose `tagcpy` and `tagmove` functions, but this brings up questions over the proper purpose of the `memcpy` and `memmove` functions in a tagged architecture. Without linearity, memory copying in Taxi affect both data and tags. We defer analysis of this arrangement to future work.

Chapter 7

Implementation of Taxi

In this chapter, we outline our implementation of the Taxi code reuse defense from Chapter 6. Taxi supports a series of tagged memory policies to protect control flow data through modifications to an existing hardware architecture, its compiler toolchain, and its operating system. We have successfully implemented Taxi and a subset of our policies on top of the open-source RISC-V architecture [8], and we outline our experiences applying those policies to a number of programs on the Linux kernel here. We also analyze the impact of the size of Taxi's tag cache on its miss rate for the SPEC2006 benchmarks [56].

7.1 Methodology

We have implemented Taxi on top of RISC-V, a small but well-supported architecture. While a version of the popular hardware emulator QEMU [9] supporting RISC-V was available, we chose to implement our policies on the Spike instruction set simulator provided by the RISC-V team. Despite the recent volatility of the RISC-V instruction set specification [64], Spike is mature enough to run a variety of software on top of the Linux kernel and is an excellent testbed for our policies. Spike also provides a cache simulation framework to evaluate miss rates; we modify it to simulate Taxi's tag cache as well.

Taxi is based upon the version of `spike` made available on GitHub on January

9th, 2015 [62], is known to run well on Ubuntu 14.04, and is available at `https://github.com/riscv-mit/`. Our modifications to the RISC-V `gcc` 4.9.2 toolchain and Linux 3.14.29 kernel are based on the versions available from February 1st and February 17th. Our version of `spike` utilizes version 2.0 of the RV64 user-level instruction set with the M, A, F, and D extensions. To test user-level programs, we use `busybox` to simulate a simple Unix-like user environment with a full file-system.

To exhaustively test the security and compatibility implications of our policies, we have gathered a large corpus of test programs. These range from system utilities like the `bash` shell and a port of the system call fuzzer Trinity [31] to the `gcc` compiler "torture" tests and the CPython interpreter. As it is difficult to exhaustively test the functionality of large programs, we also created our own test programs. Some of these test the functionality of individual policies (like preventing code reuse attacks using return addresses) but others showcase important C functionality and idioms that are broken by other defenses. In particular, we provide tests for the functionality of C idioms expressed in [17] and test the behavior of exceptions of the call and return control flow discipline expressed in [22]. We have also put considerable effort into expanding the debugging facilities in `spike` to aid understanding of how policies influence program behavior, including significant memory tracing infrastructure. To allow for tests that are aware of the presence of `spike`, we also created `libspike`, a small user library.

Despite these resources, our efforts to empirically examine Taxi and its policies face some limitations. The lack of network support in `spike` restricts us from testing our policies on vulnerable tools like OpenSSL and web servers like Apache and `nginx`. This is a significant limitation because in practice code reuse attacks are often performed in remote settings. As it is an interpreter, `spike` limits our ability to analyze the performance impacts of policies. While we can identify the increase in the number of operations that Taxi requires a processor to perform, an implementation of Taxi in an emulator or in hardware would allow for an analysis including data on TPU and tag cache latencies. Despite these limitations, our `spike` based framework still provides us with significant information on the capabilities of tagged memory policies.

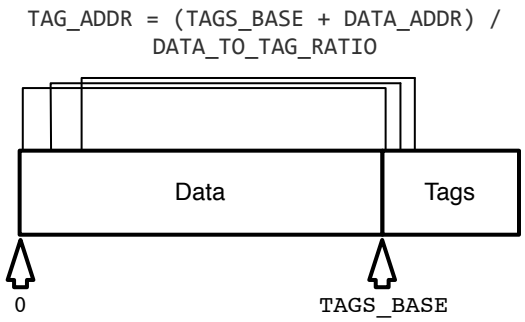


Figure 7-1: The layout of physical memory in our Taxi implementation. Because the ratio of data to tags is constant, we can easily compute the address of tags given the physical address of data.

7.2 Modifications to RISC-V

We augment memory in RISC-V to include tags through the use of shadow space. This ensures that we do not have to widen the processor word size, which would require significant modifications to the entire processor, the Linux kernel, and our compiler toolchain. To allow for 8 bit tags, we reserve the top $\frac{1}{8}$ of physical memory. We add tags to registers by creating a new set of byte registers that are accessed in conjunction with existing registers.

Read and write operations on all registers except the program counter are augmented to operate on both data and tags. Operations on memory locate tags through a new address-translation phase that determines tag-physical addresses from data-physical addresses. This scheme, depicted in Figure 7-1, is simple to implement in hardware and unlike CPI's safe region makes tags inaccessible by placing them outside of virtual memory. The security of tags in Taxi does not depend on this decision: an alternative scheme exposing tags in a read-only region of virtual memory could also ensure tag integrity. Deciding where to place tags can significantly impact program performance, but given the limitations of our implementation we do not explore this topic further. Instead, we note that address-translation schemes that determine tag and physical addresses in parallel are likely to see the best performance.

Tag propagation is implemented by applying the rules presented in Table 6.1 to each instruction in `spike`, generating traps when necessary. For processor-specific

instructions like writing to RISC-V control registers, we write zero tags; we have not noticed any side effects of this scheme.

We signal software on tag violations by creating a new "tag" hardware trap in `spike`. To recognize this trap in hardware, the Linux kernel must be made aware of it to generate a signal for the offending user program. For testing purposes, we generate the `SIGBUS` signal whose default handler exits the program. A dedicated operating system tag signal with a customizable handler would allow programs to handle tag violations on demand and safely exit otherwise, functionality not provided by the standard POSIX signals.

The changes required to the `gcc` toolchain required to support Taxi are minimal, mostly consisting of allowing new instruction opcodes to be used inside inline assembly. Supporting the policies outlined in Section 6.5 requires more extensive modifications to incorporate new static analyses and allow an initial set of program tags to be loaded in with the program binary. We leave this to future work.

Finally, to allow for per-process control of tag enforcement, we add a new processor flag to `spike` that denotes "tag mode." Tag mode is off by default but can be enabled and disabled through a new `tagenforce` instruction, allowing processes to use tag protection at will. If stored in a per-process data structure, like a RISC-V control register or the Linux process `struct`, this flag can be preserved across kernel context switches. While we have not implemented this feature, it can help reduce the effect of policy edge cases as careful programs can simply turn tag protection off when they intentionally violate policies.

7.3 Evaluation of Policies

7.3.1 Call/Return Discipline Protection

With the described modifications to `spike`, the `gcc` toolchain, and the Linux kernel, enforcing the call and return discipline requires minimal code changes. As RISC-V is a RISC architecture, it provides two jump-and-link instructions in place of dedicated

function call and return instructions. The first, `jal`, jumps to an address encoded in the instruction's immediate and saves the original incremented program counter in the destination register. This is used for direct function calls and unconditional jumps when the destination register is the dedicated zero register. The second, `jalr`, jumps to the address contained in the source register added to the provided intermediate while saving the incremented program counter similarly. This instruction is used for indirect function calls but also for returns when the destination register is the dedicated zero register. Therefore, it is easy to determine the purpose of any `jal` or `jalr` instruction and instrument it as such.

Our changes to the `jal` instruction are minimal: we simply set the destination register's tag to our return address tag. In practice, function calls use the RISC-V dedicated return address register as their destination. We do not have to do any work to avoid tagging unconditional jumps because by RISC-V convention, all such jumps save the incremented program counter to the dedicated zero register, which accepts all writes without performing any action. There is also no danger of reading a return address tag from the zero register. In total, instrumenting `jal` requires only one line of code to be modified.

Modifying the `jalr` instruction requires slightly more work, as we must apply the return address tag on indirect function calls and verify the return address tag on returns. Our instrumented `jalr` is presented in Listing 7.1. In total, instrumenting `jalr` requires adding four lines of code to verify the return address tag (lines 4-7) and modifying one line of code to set the return tag for indirect function calls (line 11). Because returns use the dedicated zero register as their destination register, we do not have to avoid tagging on returns.

We note that the RISC-V architecture does not ensure that return addresses created by `jal` and `jalr` instructions are stored in the dedicated return address register. We could choose to enforce this constraint through further modifications to the architecture and compiler toolchain, but we have not seen any situation where `gcc` generates code that stores return addresses in any other register in practice. This is not an issue for architectures with a prescribed destination for return addresses.

```
1  reg_t tmp = npc; // already incremented pc value
2  // check the tag on the jump target to make sure it is ok to execute
3  // but only if RS1 is ra, the conventional return pointer register
4  if ((TAG_ENFORCE_ON) && (!(TAG_S1 & TAG_PC))
5          && (insn.rs1() == RETURN_REGISTER) && (!IS_SUPERVISOR)) {
6      TAG_TRAP();
7  }
8
9  // write the new pc value and tag
10 set_pc((RS1 + insn.i_imm()) & ~reg_t(1));
11 WRITE_RD_AND_TAG(tmp, TAG_PC);
```

Listing 7.1: Taxi's instrumented `jalr` instruction. Here, `reg_t` represents an unsigned integral type large enough to hold any 64-bit value and `TAG_PC` is our return address tag. Line 10 sets the program counter's next value to be the sum of the source register and the instruction's encoded immediate, aligning if necessary.

Observed Security

Despite its modest changes, the call and return discipline provides a high level of security for return addresses. The buffer overflow in the vulnerable C program in Listing 1.1 indeed generates a tag trap, and our more targeted exploit in Appendix A is detected as well. We detected no false positives in our functionality test programs, which perform system calls like `fork`, run sorting algorithms, and interact with the file system. We did not receive any tag traps while running the `bash` shell, the chess program `gnuchess`, or the CPython interpreter. The system call fuzzer Trinity noted no problems interacting with the operating system.

Because Taxi does not detect or prevent buffer overflows, test suites like the Wilander test suite [66] are not appropriate to assess the security of our policies. Real-world exploits should be used as replacements for these tests and would provide considerable information for a Taxi implementation with network capabilities.

Program Compatibility

In Section 6.4.1, we noted that two problematic cases for the call and return discipline policy are Unix signal handlers and C++ exceptions. As expected, they both produced spurious traps in our implementation. To allow for correct signal handling

```
1  static int setup_rt_frame(struct ksignal *ksig, sigset_t *set,
2          struct pt_regs *regs)
3  {
4          ...
5          /* Create the ucontext. */
6          ...
7          err |= setup_sigcontext(&frame->uc.uc_mcontext, regs);
8          if (err)
9                  return -EFAULT;
10
11         /* Set up to return from userspace. */
12         regs->ra = (unsigned long)VDSO_SYMBOL(
13                 current->mm->context.vdso, rt_sigreturn);
14         ...
15 }
```

Listing 7.2: Initialization of signal handlers in the Linux kernel. Here, the return address of the signal handling context `regs` is modified to point to the address of the `sigreturn` system call.

without generating traps, we introduce a small modification to the Linux kernel. When a signal with a designated user handler is generated, the Linux kernel normally saves the existing program context and allocates a new context for signal handling. To ensure that user signal handlers return to the `sigreturn` system call and thus restore the original program context, Linux in RISC-V modifies the value of the return address register in the signal handling context. The relevant section of the RISC-V Linux 3.14.29 port [1] is presented in Listing 7.2.

This action creates a return address outside of the two function call instructions `jal` and `jalr` and is therefore a violation of our policy. Nevertheless, Taxi already trusts the kernel, so we can safely allow this violation. We trust this return address using the `settag` instruction as depicted in Listing 7.3. With this three-line addition, programs with custom signal handlers function without generating any tag traps.

C++ exceptions present a more complicated situation. In line with the modern C++ philosophy, `gcc` implements exceptions at zero cost to programs that do not use them. On RISC-V, `gcc` implements exceptions using the two-phase method developed for the Intel Itanium architecture [20]. In this method, the correct exception handler is first identified when an exception is thrown. After the stack is unwound to the

```
1       /* Set up to return from userspace. */
2       regs->ra = (unsigned long)VDSO_SYMBOL(
3         current->mm->context.vdso, rt_sigreturn);
4
5       /* Bless the tag of the new value of regs->ra. */
6       __asm__ __volatile__ ("settag %0, 1"
7         : "=r"(regs->ra)
8         : "r"(regs->ra));
9       ...
```

Listing 7.3: Trusting signal handler return addresses during initialization. We use GNU inline assembly to force the address of `sigreturn` into a register, where it can be trusted by the `settag` instruction. In this example, the value of `TAG_PC` is `1`.

appropriate position, the exception handler is run with a reference to the exception object.

The `gcc` toolchain handles this situation in its `libgcc` runtime library by starting the exception handler through a function call to the inlined `uw_install_context` function, which loads register values including the return address register and executes a return. The infrastructure that determines the right exception handler to run also controls the value of the return address register right before the return occurs, giving it power over control flow. As we trust `libgcc`, we again safely allow a technical violation of our policy by trusting the contents of the return address register inside of `uw_install_context` as illustrated in Listing 7.4. With this one line addition, programs that use C++ exceptions function without generating tag traps.

With modifications to support Unix signal handling and C++ exceptions, Taxi's call and return discipline policy is very compatible in practice, generating no tag traps on any of our benign test programs, the SPEC2006 benchmarks, the `gcc` 4.9.2 torture tests, or during our limited experiences with larger programs like CPython.

7.3.2 Linearity of Return Addresses

Enforcing return address linearity in `spike` requires a set of tag propagation rules that filter out return address tags whenever possible. We provide this by implementing the modified propagation rules presented in Table 6.2. Linearity also requires that return

```
uw_install_context(&this_context, &cur_context)
{
        addi        sp, sp, 736
        ld          ra, 2024(sp)
        ld          s0, 2016(sp)
        ...
        fld         fs11, 1792(sp)
        settag      ra, 1
        addi        sp, sp, 2032
        ret
}
```

Listing 7.4: Trusting C++ exception handler addresses in `libgcc`. Here, in RISC-V assembly, we can see that the values of registers are being loaded from a context on the stack. As `libgcc` is trusted, we trust the value of the return address register before it is used, allowing the execution of a C++ exception handler. Note that `ret` is RISC-V pseudo-assembler for `jalr ra`.

address tags are *moved* when return addresses are copied. By prohibiting arithmetic on return addresses, we only need to instrument memory loads and stores.

To prevent memory loads from creating multiple copies of return addresses with the correct tag, we instrument all RISC-V load instructions to check for return address tags. When a return address tag is found, load instructions must mask it out and write the resulting tag back to memory. Note that because memory accesses can cause page faults, load instructions must be re-entrant. Our instrumented RISC-V double-word load instruction `ld` is presented in Listing 7.5. Load instructions for smaller data sizes are instrumented in a similar way.

Likewise, memory stores must care to mask out return address tags from the origin of data they store. In RISC-V this extra store operation is to a register, permitting lower overhead for this operation. We present an instrumented re-entrant version of the RISC-V double-word store instruction `sd` in Listing 7.6. Store instructions for smaller data sizes are handled similarly.

Observed Security

As enforcing linearity of return address tags is only meaningful in the presence of the call and return discipline policy, linearity in practice provides security at least as

```
1  reg_t addr = RS1 + insn.i_imm();
2  tagged_reg_t v = MMU.load_tagged_int64(addr);
3
4  if ((TAG_ENFORCE_ON) && (!IS_SUPERVISOR)) {
5    // Clear TAG_PC from the memory location if present.
6    tag_t cleared_tag = CLEAR_PC_TAG(v.tag);
7    if (cleared_tag != v.tag) {
8      MMU.store_tag_value(cleared_tag, addr);
9    }
10 }
11
12 WRITE_RD_AND_TAG(v.val, v.tag);
```

Listing 7.5: Taxi's instrumented `ld` instruction. In this example, `tag_t` represents an unsigned integral type capable of holding tags. To make sure that loading tagged return addresses does not create copies of return address tags, on Line 8 we store a sanitized tag back to the source address if it previously was a return address.

strong as that policy. Experiments have confirmed that our linearity implementation provides this guarantee, protecting no fewer programs from code reuse attacks using return addresses than our call and return policy implementation. Taxi also detects our own return address replay attack (available in Appendix A) without inducing tag traps in `bash`, Trinity, SPEC2006, or any other of our own test programs.

The `gcc` torture tests provided one interesting false positive: the `pr47237` test uses the `gcc __builtin_apply` C function to transfer control to a function without explicitly calling it. This built-in function erroneously performs a load instruction on the return address of the program's `main` function without a matching store, stripping its tag. Because the copy of the return address is not used, this copy is spurious and could be eliminated without breaking correctness. Indeed, we did not see this problem when compiling with the `gcc` optimizer.

Program Compatibility

We note in Section 6.4.2 that only programs that copy or modify return addresses willingly are likely to violate linearity. This includes programs that use the C `setjmp` and `longjmp` functions for non-linear control flow, as those functions save and restore program state (including return addresses) from the `jmpbuf` data structure. Because

```
reg_t addr = RS1 + insn.s_imm();
MMU.store_tagged_uint64(addr, RS2, TAG_S2);

// Ensure we have only one live PC tag by clearing the PC tag
// from the source register.
if ((TAG_ENFORCE_ON) && (!IS_SUPERVISOR) &&
        (insn.rs2() == RETURN_REGISTER)) {
  CLEAR_TAG(RETURN_REGISTER, TAG_PC);
}
```

Listing 7.6: Taxi's instrumented `sd` instruction. After storing data from the return address register, we strip any return address tags left in the register.

register configuration is architecture-dependent, `setjmp` and `longjmp` require the use of architecture-specific assembly despite being part of the C standard. This allows us to introduce modifications without breaking language standards compliance.

The `setjmp` function saves the current execution context into a `jmpbuf` data structure that can be restored by calling `longjmp`. When a `jmpbuf` context is restored, control flow resumes directly after the `setjmp` instruction where the program's context was saved. The actual behavior of `setjmp` depends on whether it is inlined. When it is, the return address copied into the `jmpbuf` is from the parent function. When `setjmp` is not inlined, the return address it copies refers to the instruction after the call to `setjmp`. In either case, the return address must be trusted before `setjmp` returns to avoid generating a spurious trap. In the inline case, this creates a small vulnerability as the return address of the parent may be controlled by an attacker. This can be avoided by using a load tag instruction or similar mechanism to determine the return address tag at runtime before blessing it. On RISC-V `setjmp` is not inlined, so we do not need to implement such a mechanism. The program in Listing 7.7 helps illustrate how `setjmp` affects the linearity policy implementation.

The `longjmp` function also requires a small modification. When a user calls `longjmp` to restore a previous program context, the return address inside the context is copied into the return address register, stripping the original of its tag under linearity. To be able to use a `jmpbuf` context multiple times, we trust its return address after loading it. If the value of this return address is controlled by an attacker, we

```
1  int useful_function(int important_number) {
2    jmp_buf context;
3    if (setjmp(context))
4      return -1;
5
6    if (important_number == 42)
7      longjmp(context, 1);
8    return 0;
9  }
10
11 void parent(void) {
12   useful_function(42);
13   useful_function(0);
14 }
```

Listing 7.7: An example of how return address linearity affects `setjmp`. If `setjmp` is inlined into `useful_function`, the return address it will copy into `context` on its first execution will be the address of line 13. Naively blessing this return address can allow attackers to gain control over the program counter. When `setjmp` is not inlined, it will store the address of line 6.

will detect it as the original tag's value is inspected when `longjmp` returns.

Finally, we note that if applied to the kernel, the linearity policy would be incompatible with `fork` and related system calls. In Linux, when a process calls `fork` its memory contents are duplicated into the new process. With linearity, this copy operation strips the tag of all return addresses in the original process. One way of maintaining linearity while allowing the kernel to perform `fork` would be creating a new privileged tag copying instruction. Because we have limited tag policies to user space this is not an issue in Taxi.

7.3.3 Restricting Partial Copies

The partial copy policy aims to protect return addresses further by not preserving their tags on partial memory stores, eliminating attacks that carefully store only part of a return address to acquire its tag. We implement this policy in Taxi by masking out return address tags in all instructions that store data smaller than the word size into memory. In 64-bit RISC-V there are six instructions that fit this description, requiring in total only six lines of instrumentation code. A detailed analysis of the

practical security and compatibility of restricting partial copies of return addresses is provided in [28].

7.3.4 Data Blacklisting

By applying the observation that return addresses are usually stored and loaded atomically to all pointers, the blacklist policy prevents pointers that are corrupted during buffer overflows from being stored to, loaded from, or jumped to. This protects programs from pointers corrupted by C functions like `sprintf` and `strcpy`, which are vulnerable to buffer overflows. In Taxi, blacklisting is enforced by creating a new tag, `TAG_DATA`, and applying it to the target of all store instructions that operate on memory smaller than the word size. The `jalr` instruction is instrumented to prevent jumping to memory tagged with `TAG_DATA`. All load and store instructions that attempt to use userspace memory with this tag as addresses generate tag traps. Finally, a new set of tag propagation rules proliferate `TAG_DATA` on arithmetic instructions.

Functions that are designed to operate byte-by-byte on generic data pose problems for blacklisting. Two examples are the `memcpy` and `memmove` functions which are commonly used to copy `struct`s that hold pointers. On RISC-V, only `memmove` copies data at byte granularity; because `spike` simulates an in-order processor, we simply turn off tag enforcement for the duration of this function.

Pointers stored in memory not aligned to the processor word size can share tag bits with non-pointers in Taxi, which poses a problem when blacklisting. We have only seen this situation in the `960117-1` `gcc` torture test case when compiled without optimizations. This failure highlights the limitations of imposing tag structure on unordered memory but can be largely overcome by optimizing compilers that align data. We refer the reader to [28] for further analysis of the security and compatibility of the Taxi blacklisting implementation.

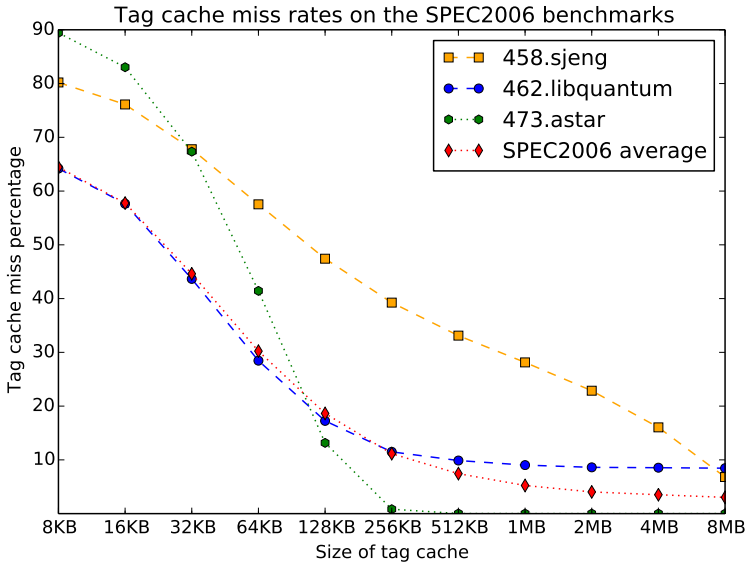
7.4 Cache Simulations

Throughout this chapter, we have shown that tagged memory policies are generally applicable to real-world programs with a host of exceptions. To show that tagged memory itself can be practical for real-world programs, we have performed simulations of the workload of Taxi's tag cache at various sizes on the SPEC2006 benchmarks. All simulated tag caches are four-way set associative, use 64-byte blocks, and use a random replacement policy. We present our findings in Figure 7-2 and Table 7.1, including results from a simulated 256KB L2 cache for comparison.

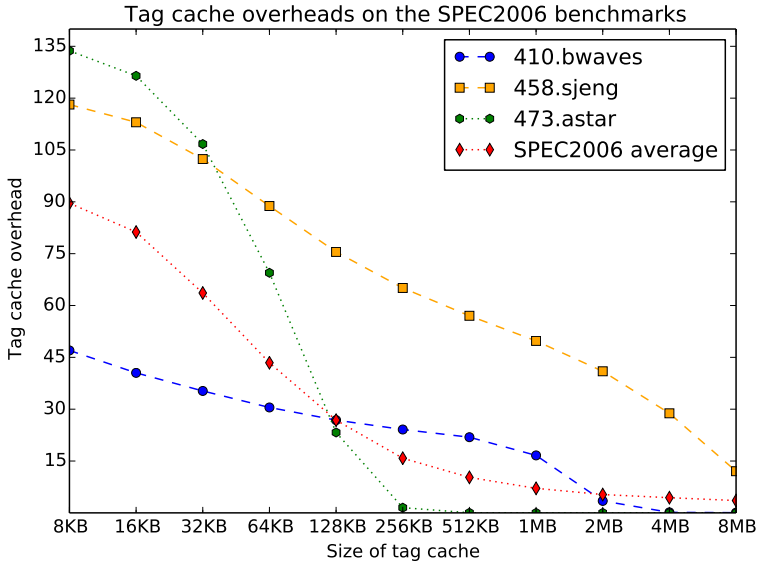
As depicted in Figure 6-2, Taxi relies on a new tag cache to handle misses on tags in the expanded L2 cache. To understand the role that this tag cache plays in our memory hierarchy, we simulated 11 tag caches in `spike` of different sizes, observing both the tag miss rate and the number of evicted tags written back to memory. We compactly summarize these two measurements in our notion of "total overhead", which we define as $\frac{cache\ misses\ +\ writebacks}{tag\ cache\ accesses}$. Because the number of writebacks is also a function of the number of hits to the tag cache, this number can be over 100%. Figure 7-2 presents graphs of this overhead and tag miss rate for a subset of the SPEC2006 benchmarks.

Initially, we reasoned that processors with large data to tag ratios would require only small tag caches. Figure 7-2a shows that in our Taxi implementation with 64 bytes of data for each tag, tag caches smaller than 256KB have high miss rates across the board. Benchmarks with smaller working sets like `473.astar` have near zero miss rates at this size, while our average miss rate is just over 11%. As the tag cache grows, this rate does not improve for most benchmarks, suggesting that compulsory and conflict cache misses dominate in larger caches. Future analyses could test this claim by varying replacement policies.

Without precise knowledge of the latency of tag cache misses, it is hard to say exactly what constitutes a good miss rate. To help answer this question, we can compare tag and L2 cache miss rates. While the average miss rate of a 256KB tag cache for SPEC2006 is 11.15%, Taxi's 256KB L2 cache achieves an average miss rate



(a) Tag cache miss rates by tag size for three benchmarks. Small cache sizes exhibit high miss rates for all benchmarks. Miss rates for benchmarks like `458.sjeng` never level off, possibly indicating large working sets. Most benchmarks fare better but do not improve significantly after 256KB; benchmarks like `473.astar` are best case scenarios.



(b) Total tag cache overhead by tag size for a different set of benchmarks. As tag cache overhead includes both cache misses and data writebacks, it can exceed 100%. Tag caches smaller than 128KB exhibit high overheads for all benchmarks, but average overhead is only about 10% at 512KB.

Figure 7-2: Simulated tag cache miss rates and total overhead for select SPEC2006 benchmarks.

Benchmark	**8KB**	**64**	**256**	**512**	**1MB**	**4MB**	**512/8KB**	**L2**
400.perlbench	66.13	22.72	3.46	1.82	1.19	0.84	2.76	2.42
401.bzip2	90.78	53.39	1.02	0.09	0.02	0.01	0.09	37.54
403.gcc	63.05	26.31	5.66	1.93	0.36	0.22	3.07	5.33
410.bwaves	38.25	25.66	21.10	19.29	14.40	0.10	50.43	4.88
416.gamess	60.74	25.39	8.25	6.54	5.84	5.40	10.77	6.06
429.mcf	77.24	34.60	8.52	4.96	3.82	2.85	6.42	32.37
433.milc	71.68	54.21	12.91	0.12	0.01	0.00	0.16	40.79
435.gromacs	58.91	15.24	5.00	2.27	1.28	1.00	3.85	2.37
436.cactusADM	58.72	21.62	11.98	8.68	0.78	0.19	14.78	9.32
437.leslie3d	59.83	24.62	5.62	0.91	0.01	0.00	1.52	9.56
444.namd	56.47	23.18	9.39	6.00	3.53	1.50	10.62	0.41
445.gobmk	67.71	34.08	13.45	8.54	3.93	0.05	12.61	2.80
447.dealII	49.42	24.48	11.28	6.74	2.11	0.37	13.65	2.72
450.soplex	64.70	21.09	8.08	6.64	6.19	5.90	10.26	3.64
454.calculix	62.38	22.95	8.67	7.04	6.39	6.06	11.29	3.62
456.hmmer	63.76	25.46	10.06	8.56	7.60	7.03	13.43	0.07
458.sjeng	80.20	57.53	39.25	33.11	28.13	16.05	41.28	3.35
459.GemsFDTD	71.92	48.36	25.49	17.43	13.58	12.31	24.24	29.16
462.libquantum	64.19	28.45	11.49	9.88	9.03	8.54	15.39	0.72
464.h264ref	61.85	32.15	5.44	0.80	0.16	0.05	1.30	10.00
465.tonto	62.23	15.49	4.46	3.60	3.21	2.99	5.79	0.55
470.lbm	58.51	24.28	13.33	9.00	7.09	6.32	15.38	13.38
473.astar	89.43	41.44	0.82	0.02	0.01	0.01	0.02	10.72
481.wrf	55.67	20.98	10.52	7.70	4.66	0.79	13.83	7.10
482.sphinx3	52.32	34.88	25.38	12.97	2.24	0.25	24.79	21.25
483.xalancbmk	70.59	30.97	7.87	3.50	2.70	2.39	4.96	2.84
998.specrand	63.88	28.21	11.75	9.96	9.04	8.60	15.59	5.19
999.specrand	63.76	28.60	12.02	10.12	9.23	8.58	15.87	5.17
Average	64.44	30.23	11.15	7.44	5.23	3.51	12.29	9.76
Standard dev.	10.76	11.03	8.02	6.90	5.92	4.23	11.53	11.33

Table 7.1: Simulated tag cache miss rate (out of 100) by size for the SPEC2006 benchmarks. All tag caches are four-way associative with 64-byte blocks. For comparison, we list the miss rate of the 256KB, eight-way associative L2 cache with 128-byte blocks. Note that the L2 cache has a higher miss rate than tag caches 512KB or larger. On `spike`, the `434.zeusmp`, `453.povray` and `471.omnetpp` benchmarks were non-functional without regard to tag policy.

of 9.76%, only about 1.4% lower. This is encouraging because caches further down in memory naturally tend to have higher cache miss rates even at large sizes. A 512KB tag cache is enough to beat the L2 miss rate, suggesting that large data to tag ratios do play some role in lowering miss rates. This analysis may also extend to writebacks, which we have observed to behave similarly in the SPEC benchmarks.

We conclude our analysis by noting that the SPEC benchmarks are designed to test processor performance, and are not a perfect tool to analyze memory hierarchy. There also does not seem to be a way of predicting how benchmarks should perform. This is partly because of the lack of a language feature that carries an obvious performance penalty like C++ virtual functions do for CPI. Nevertheless, the low standard deviation of miss rates on the SPEC benchmarks is encouraging and provides a clear standard for future tag cache analyses to reproduce.

Chapter 8

Future Work

The four tag policies explored in this thesis only begin to harness the expressiveness of policies on tagged memory. The policies outlined in Section 6.5 represent starting points for further analyses of tagged memory, but all require a tag-aware compiler to generate a set of starting tags for a program that are loaded into memory on execution. We have not yet explored the modifications required for a production compiler like `gcc` or `llvm` to support tagged memory, but extensive data flow analyses that can function in the presence of casting are likely necessary to determine when to manipulate tags. Even without these analyses, tag-aware compilers can assist in policy implementation and protect important static data structures like the Global Offset Table. The capabilities of tag-aware compilers to implement policies without a security focus also represent an avenue of research.

The blacklist policy could be significantly strengthened when paired with a tag-aware compiler, which through analysis or annotations could identify buffers with contents that are never used as pointers. This would greatly expand the availability of protection currently offered to data manipulated by C string functions, especially for often vulnerable web servers that must read data over a socket into a buffer. The overhead of this implementation would be minimized by statically instrumenting `libc` but could also be practical with hardware support for efficient assignment of tags to large regions of memory. This support would also be useful for quickly loading an initial set of tags into memory on program startup.

The Taxi implementation discussed in this thesis is capable of demonstrating the practical security and compatibility guarantees of tagged memory policies but is otherwise limited in its ability to simulate a real processor. A Taxi implementation on top of a full processor emulator like QEMU [9] would be much more capable of determining properties of hardware design like the TPU's latency. Examination of the impact on tag cache performance of properties like set associativity and block size could be performed. Such an implementation would also be capable of a full, reportable run on the SPEC2006 benchmark suite where program runtime overhead could be measured and compared with the overhead of defenses like Code Pointer Integrity [34].

Existing code reuse defenses could be strengthened with hardware tag policies. For instance, CPI could strengthen its isolation of code pointers by marking them in the initial set of tags loaded at program initialization. These tagged code pointers would be protected from buffer overflows if their tags could not be forged by attackers. While more complex than memory segmentation, a correct implementation would be immune from attacks like the one presented in Chapter 4 and would be more robust to information disclosure attacks. An implementation of Control Flow Integrity [4] on a tagged architecture like Taxi could use a larger number of labels to distinguish types of control flow transfers and enhance its security guarantees.

Taxi's use of limited tagged memory makes construction of policies that prevent other memory corruption attack vectors difficult. Proposed solutions for preventing temporal memory vulnerabilities like the lock and key strategy from CETS [39] do not translate well to Taxi, as they require use of very large counter values. This does not prevent use of Taxi as a component of a software-based memory safety defense.

Although our proposed policies only explore tagging data, Taxi in principle is also capable of supporting policies that use tags on instructions. This allows multiple versions of instructions differentiated only by tag values, further expanding the expressiveness of tag policies and the architecture itself. We hope that this expressiveness will foster the design and implementation of additional tagged memory policies that can prevent broad classes of memory corruption attacks.

Appendix A

Sample Test Programs

```
1  void test() {
2      int x[10];
3
4      x[12] = (0xdeadbeef); // Overwrite our return address.
5  }
6
7  int main(void) {
8      __asm__ __volatile__("tagenforce ra,1");
9      test();
10
11     return 0; // This should not be reached.
12 }
```

Listing A.1: A simple buffer overflow test program, where the return address is overwritten to an invalid value. Without Taxi, this program is likely to terminate with a segmentation fault. Under Taxi's call and return discipline policy, this program will instead receive a tag trap signal.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void b(int arg1) {
5    printf("Calling harmless function b() with argument
6      %x", arg1);
7  }
8
9  void a(int arg1) {
10   int k = arg1 & 0x0f0f0f0f;
11   b(0xaaaaaaaa);
12   if(k != 0x0e0d0e0f) {
13     abort();
14   }
15 }
16
17 void do_stuff(int arg1) {
18   int local = 0;
19   a(arg1);
20
21   // Simulate a replay attack on function a
22   int *p = &local;
23   // Setup the return address to return into a.
24   p[5] = p[-19];
25   // Setup the argument too
26   p[2] = 0x87654321;
27 }
28
29 int main(void) {
30   do_stuff(0xdeadbeef);
31
32   return 0;
33 }
```

Listing A.2: A program with a return address replay attack. Here, `do_stuff` calls `a` which in turn calls `b`. After `a` returns to `do_stuff`, `do_stuff` overwrites its own return address through pointer arithmetic to return to `a`. Without Taxi (on a system with a compatible stack layout), this program will execute `a` twice and abort. With Taxi's return address linearity policy enforced, the return address altered in line 24 will not have the correct tag and a tag trap will occur when `do_stuff` returns.

Bibliography

[1] Linux kernel for RISC-V. `https://github.com/riscv/riscv-linux`, 2012. Accessed: 2015-07-30.

[2] CVE-2014-0160. Available from MITRE, CVE-ID CVE-2014-0160, December 3 2013. Accessed: 2015-07-06.

[3] Intel MPX Support in the GCC Compiler. https://gcc.gnu.org/wiki/Intel MPX support in the GCC compiler, 2015. Accessed: 2015-07-06.

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

[5] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 51–66, Berkeley, CA, USA, 2009. USENIX Association.

[6] Jim Alves-Foss, Jia Song, Stuart Steiner, and Saeede Zakeri. A new operating system for security tagged architecture hardware in support of multiple independent levels of security (mils) compliant system. Technical report, University of Idaho, 2014.

[7] James P Anderson. Computer security technology planning study. volume 2. Technical report, DTIC Document, 1972.

[8] Krste Asanović and David A. Patterson. Instruction sets should be free: The case for RISC-V. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley, Aug 2014.

[9] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[10] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. ACM.

[11] Erik Bosman and Herbert Bos. Framing signals - a return to portable shellcode. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 243–258, Washington, DC, USA, 2014. IEEE Computer Society.

[12] Alex Bradbury, Gavin Ferris, and Robert Mullins. Tagged memory and minion cores in the LowRISC SoC. lowRISC-MEMO 2014-001, Computer Laboratory, University of Cambridge, December 2014.

[13] Nicholas Carlini and David Wagner. ROP is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, San Diego, CA, August 2014. USENIX Association.

[14] Hong Kei Chan. A technical analysis of CVE-2014-1776. `http://blog.fortinet.com/post/a-technical-analysis-of-cve-2014-1776`, 2014. Accessed: 2015-07-06.

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

[16] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[17] 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. *SIGARCH Comput. Archit. News*, 43(1):117–130, March 2015.

[18] IBM Corporation. IBM system i. `http://www-03.ibm.com/systems/i`. Accessed: 2015-07-16.

[19] Intel Corporation. Intel i960 processors. `http://www.intel.com/design/i960/`. Accessed: 2015-07-16.

[20] Intel Corporation. Itanium C++ ABI: Exception handling. `http://mentorembedded.github.io/cxx-abi/abi-eh.html`, 1999. Accessed: 2015-07-30.

[21] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, August 2014. USENIX Association.

[22] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings*

of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS '11, pages 40–51, New York, NY, USA, 2011. ACM.

[23] 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.

[24] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and Andre DeHon. Architectural support for software-defined metadata processing. *SIGARCH Comput. Archit. News*, 43(1):487–502, March 2015.

[25] C. Baker et al. The Symbolics Ivory processor: a 40 bit tagged architecture Lisp microprocessor. In *Proceedings of the 1987 IEEE International Conference on Computer Design*, pages 512–515, October 1987.

[26] Isaac Evans, Sam Fingeret, Julián González, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *Proc. of IEEE S&P*, 2015.

[27] Edward A. Feustel. On the advantages of tagged architecture. *IEEE Trans. Comput.*, 22(7):644–656, July 1973.

[28] Samuel Fingeret. *Defeating Code Reuse Attacks with Minimal Tagged Architecture*. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2015. To appear in September 2015.

[29] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 575–589, Washington, DC, USA, 2014. IEEE Computer Society.

[30] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[31] Dave Jones. Trinity: Linux system call fuzzer. `https://github.com/kernelslacker/trinity`, 2011. Accessed: 2015-07-30.

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

[33] T.F. Knight. *Implementation of a List Processing Machine*. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1979.

[34] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, Broomfield, CO, October 2014. USENIX Association.

[35] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, San Diego, California, June 2007.

[36] Ali José Mashtizadeh, Andrea Bittau, David Mazières, and Dan Boneh. Cryptographically enforced control flow integrity. *CoRR*, abs/1408.1451, 2014.

[37] Nicholas D. Matsakis and Felix S. Klock, II. The Rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104, New York, NY, USA, 2014. ACM.

[38] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. Opaque control-flow integrity. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS)*, San Diego, California, February 2015.

[39] 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.

[40] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for C. *SIGPLAN Not.*, 44(6):245–258, June 2009.

[41] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, May 2005.

[42] Tim Newsham. Format string attacks. `http://www.thenewsh.com/~newsham/format-string-attacks.pdf`, 2000. Accessed: 2015-07-06.

[43] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996. Accessed: 2015-07-06.

[44] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 447–462, Berkeley, CA, USA, 2013. USENIX Association.

[45] Debian Project. Debian security advisory: Dsa-2896-1 openssl – security update. https://www.debian.org/security/2014/dsa-2896, 2013. Accessed: 2015-07-06.

[46] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. Program. Lang. Syst.*, 17(2):233–263, March 1995.

[47] 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.

[48] 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, *Research in Attacks, Intrusions and Defenses*, volume 8688 of *Lecture Notes in Computer Science*, pages 88–108. Springer International Publishing, 2014.

[49] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[50] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[51] Ryota Shioya, Daewung Kim, Kazuo Horio, Masahiro Goshima, and Shuichi Sakai. Low-overhead architecture for security tag. *IEICE Transactions*, 94-D(1):69–78, 2011.

[52] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 574–588, Washington, DC, USA, 2013. IEEE Computer Society.

[53] Eric Soderstrom. Analysis of return oriented programming and countermeasures. Master's thesis, Massachusetts Institue of Technology, Cambridge, MA, 2014.

[54] SPEC. Standard performance evaluation corporation benchmarks. http://www.spec.org/osg/cpu95, 1995. Accessed: 2015-07-06.

[55] SPEC. Standard performance evaluation corporation benchmarks. http://www.spec.org/osg/cpu2000, 2000. Accessed: 2015-07-06.

[56] SPEC. Standard performance evaluation corporation benchmarks. `http://www.spec.org/osg/cpu2006`, 2006. Accessed: 2015-07-12.

[57] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *SIGARCH Comput. Archit. News*, 32(5):85–96, October 2004.

[58] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society.

[59] Ady Tal. Intel software development emulator. `https://software.intel.com/en-us/articles/intel-software-development-emulator`, 2012. Accessed: 2015-07-06.

[60] PaX Team. Pax non-executable pages. `http://pax.grsecurity.net/docs/noexec.txt`. Accessed: 2015-07-06.

[61] PaX Team. Pax address space layout randomization. `http://pax.grsecurity.net/docs/aslr.txt`, 2003. Accessed: 2015-07-06.

[62] RISC-V Team. Risc-v instruction set simulator. `https://github.com/riscv/riscv-isa-sim`, 2011. Accessed: 2015-07-30.

[63] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, San Diego, CA, August 2014. USENIX Association.

[64] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste AsanoviÄĞ. 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.

[65] Robert N. M. Watson, Jonathan Woodruff, Peter G Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[66] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. RIPE: Runtime intrusion prevention evaluator. In *In Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC*. ACM, 2011.

[67] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 225–240, Berkeley, CA, USA, 2008. USENIX Association.

[68] 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 *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 559–573, Washington, DC, USA, 2013. IEEE Computer Society.

[69] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 337–352, Washington, D.C., 2013. USENIX.