P-TAXI: Enforcing Memory Safety with Programmable Tagged Architecture

by

Witchakorn Kamolpornwijit

S.B., Massachusetts Institute of Technology (2015)

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

June 2016

Author .....
Department of Electrical Engineering and Computer Science
May 12, 2016

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

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

P-TAXI: Enforcing Memory Safety with Programmable Tagged Architecture

by

Witchakorn Kamolpornwijit

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Buffer overflow is a well-known problem that remains a threat to software security. With the advancement of code-reuse attacks and return-oriented programming (ROP), it becomes problematic to protect a program from being compromised. Several defenses have been developed in an attempt to defeat code-reuse attacks. However, there is still no solution that provides complete protection with low overhead.

In this thesis, we improved TAXI [1]–[3], a ROP defense technique that utilizes a tagged architecture to prevent memory violations. Inspired by Programmable Unit for Metadata Processing (PUMP) [4], we modified TAXI so that enforcement policies can be programmed by user-level code and called it P-TAXI (Programmable TAXI). We demonstrated that, by using P-TAXI, we were able to enforce memory safety policies, including return address protection, stack garbage collection, and memory compartmentalization. In addition, we showed that P-TAXI can be used for debugging and taint tracking.

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

Acknowledgments

This thesis would not have been possible without support, insight, guidance, and thoughtful commentary from Dr. Howard Shrobe, my thesis supervisor. I would like to give a special thanks to Julián González and Samuel Fingeret for answering many questions about TAXI and guiding me through its code base.

I would like to thank Stelios Sidiroglou-Douskos, Hamed Okhravi, Andre DeHon, Isaac Evans, participants in the Micropolicy discussion, and members of Cybersecurity at CSAIL for helping me in multiple ways related to this thesis.

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

Contents

List of Figures

List of Tables

List of Code Listings

Chapter 1

Introduction

Buffer overflow is one of the well-known types of software bugs that allows an attacker to execute malicious code by intentionally crafting a sequence of inputs to a system that overruns the boundaries of a buffer. Discovered at least as early as 1972 [5], buffer overflow is still considered one of the top three categories of the most dangerous software errors, according to MITRE [6].

Because of the severity of buffer overflow attacks, many defense mechanisms have been developed over time to protect computer systems from being vulnerable. However, new techniques of buffer overflow attacks, including code-reuse attacks and return-oriented programming, have rendered ineffective many types of defenses, including Data Execution Prevention and StackGuard. As a result, it is much more difficult to protect a program from being compromised. Several defenses have been developed in an attempt to defeat code-reuse attacks. However, there is still no solution that provides complete protection with low overhead.

Chapter 2 provides background on the history and example of buffer overflow and code-reuse attacks. In Chapter 3, we discuss in detail about Return-Oriented Programming (ROP), its variants and countermeasures.

Chapter 4 and Chapter 5 describe the tagged architecture and the RISC-V instruction set, respectively, as these core components were used to develop TAXI. In Chapter 6, we discuss TAXI, a tagged architecture approach to prevent memory

violations.

P-TAXI, which is the main contribution of this thesis, is described in Chapter 7, with the basic sets of policies shown in Chapter 8. In Chapter 9, we evaluate P-TAXI and provide ideas for future works.

Chapter 2

Background

In this chapter, we will discuss major classes of buffer overflow attacks and several countermeasures that have been developed over the past decades to prevent such attack schemes. We will also review the effectiveness of such countermeasures.

2.1 Code-Injection Attacks

Code-injection attacks are one of the simplest forms of buffer overflow attacks. To initiate the attack, an attacker would craft a payload, called "shell code," which contains machine instructions that the attacker wishes to execute. The attacker then combines the shell code with a long sequence of inputs that overflows the stack of the application, resulting in replacement of the return address on the stack with the memory address of the shell code. Alternatively, if a program utilizes the function pointer feature of the C programming language, it is also possible to override a function pointer in a vulnerable program to replace the value of the function pointer with the memory address of the shell code [7].

2.1.1 Example

The C program in Listing 2.1 receives an input and stores it in the variable `name` and displays it back to the user. If a user inputs "ABCD", the call stack of the program,

after the `gets` function in line 4 is called, will be as in Figure 2-1 [8].

```
1 #include <stdio.h>
2 void f() {
3     char name[6];
4     gets(name);
5     printf("Your name is %s\n", name);
6 }
7 void main() {
8     f();
9     return 0;
10 }
```

Listing 2.1: Example C program with a buffer overflow vulnerability.

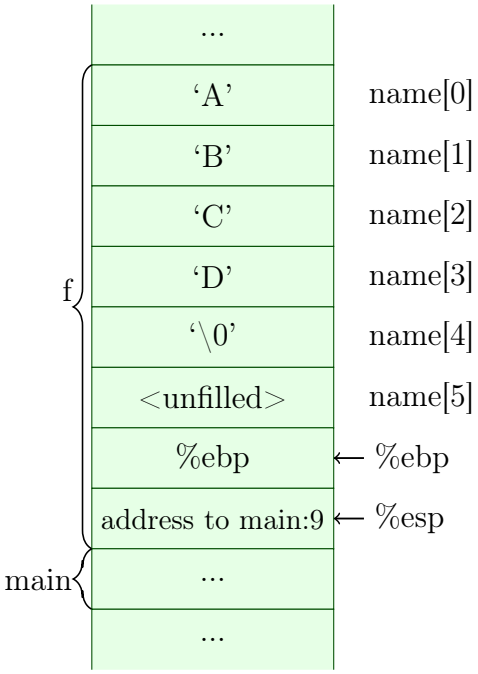

Figure 2-1: Stack after the execution of line 4 with input "ABCD".

However, we can see that in line 4, this program uses the `gets` function which does not provide a way to specify the maximum number of characters that should be read into the variable `name`. As a result, if a user of the program enters an input with more than 5 characters (which results in more than 6 bytes written to the buffer because a string has to end with a null character), the user would be able to overflow the buffer.

For example, with an input of "AAAAAAAAAAAAAA", the call stack will be as in Figure 2-2. When Function `f` is completed, the instruction pointer of the machine will be changed to 0x41414141 instead of an address that point back into function `main`.

(0x41 is an ASCII value of 'A'.) If the attacker changes the return address to point to the location of `name[0]` and puts shell code at the beginning of the input stream, the attacker would be able to execute arbitrary machine code with this program privilege.

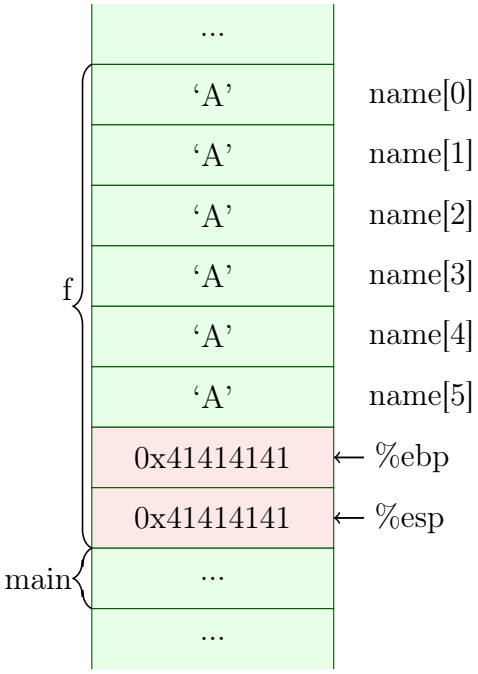

Figure 2-2: Stack after the execution of line 4 with input "AAAAAAAAAAAAA".

2.1.2 Defenses

2.1.2.1 Data Execution Prevention (DEP)

Data Execution Prevention (DEP, also known as W⊕X) is a way to prevent code-injection attacks by blocking execution of instructions that appear on marked memory regions. DEP utilizes the No-eXecute (NX) bit feature, supported by modern processors, allowing an operating system to mark certain areas of the memory as non-executable. Normally, the operating system will mark stack and heap areas of a program as non-executable, preventing attackers from providing arbitrary shell code to be executed by the program. DEP has been implemented and supported by major operating systems, including Windows, Linux, and OS X [9]–[11].

2.1.2.2 StackGuard

StackGuard (also known as stack canaries) [12] is another technique to prevent code-injection attack. To restrict modification of return addresses, the compiler is modified to add instructions to push a "canary word" to the call stack every time a return address is pushed on the stack, as shown in Figure 2-3. Instructions to check for a correct canary word are also added so that if the canary word on the stack is overwritten with an incorrect value, the program will crash instead of return. Canary words are generated randomly every time a program starts so that an attacker would not be able to predict canary words by looking at the program's binary. Many compilers have added a feature similar to StackGuard. For example, in GCC, a stack protection feature was added in version 4.1 and has been enabled by default. Users who do not want to have such protection in their program need to disable it with an argument `-fno-stack-protector` when compiling the program [13].

Since StackGuard only modifies parts of the calling convention that most programs do not rely on, there is no need to modify the source code of most programs. Operating system support is also unnecessary as the implementation can be done solely on the user level [12].

Unfortunately, StackGuard is ineffective in preventing buffer overflow attacks in multiple ways. First, StackGuard does not prevent an attacker from modifying local variables on the stack, including function pointers. In a vulnerable function that utilizes function pointers, the attacker would use the buffer overflow technique to modify the function pointers to point to the location of shell code that the attacker provided. Second, StackGuard assumes that the attacker would not be able to read the canary words. If the attacker can find a way to read canary words from the call stack, the attacker can still proceed to craft an input sequence with known canary words, thereby avoiding detection of stack inconsistency [14].

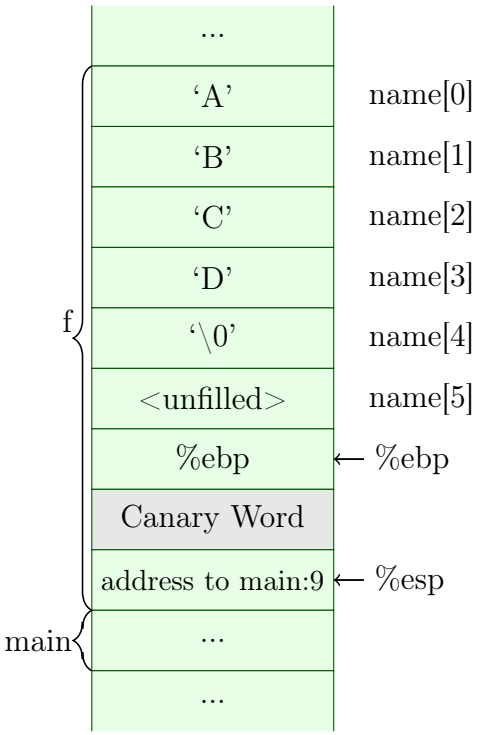

Figure 2-3: Stack in the same condition as in figure 2-1 but with StackGuard enabled.

2.2 Code-Reuse Attack

Code-reuse attacks are another form of buffer overflow attacks. In this attack scheme, instead of providing shell code directly, the attacker modifies a return address on the stack or a function pointer to point to existing code anywhere in memory. Because it is not necessary for the attacker to provide shell code as an input, the attack renders ineffective many defenses that prohibit execution of memory regions that contain program data and user input, including Data Execution Prevention [15].

However, this type of attack requires an attacker to have information about existing instructions in the regions of the memory that are executable. Therefore, compared with code-injection attacks, code-reuse attacks are likely to be more difficult to exploit on an application to whose binaries the attacker does not have access to [15].

2.2.1 Return-To-Libc Attack

A return-to-libc attack is a basic form of code-reuse attacks. By using a buffer overflow to override a return address on the stack to point to a function in libc (or another standard C library), the attacker can force a program to make a call to any function in libc, including `system`, which is a function that allow an application to run any shell command. Figure 2-4 shows an example of the call stack of the program in Listing 2.1 using the input that utilizes a return-to-lib-c attack to execute an arbitrary command by forcing a call to the `system` function in the standard C library [16].

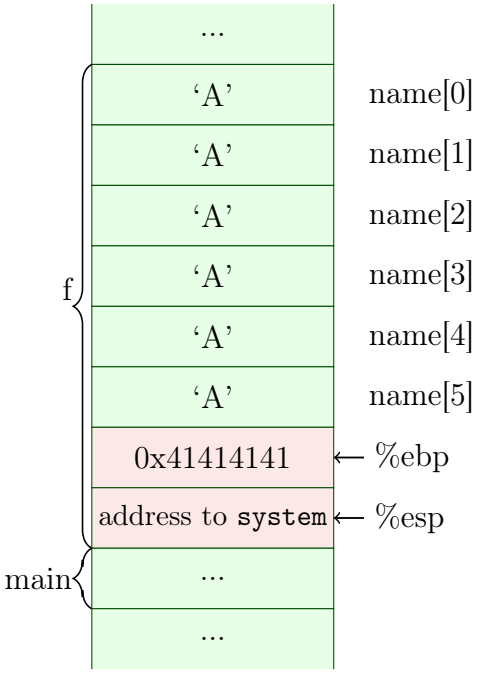

Figure 2-4: Stack after the execution of line 4 with input "AAAAAAAAAA<address to `system`>".

2.2.2 Return-Oriented Programming

Return-oriented programming (ROP) is a more advanced form of code-reuse attacks. Since there are many variations of ROP and multiple defenses that have been created in an attempt to defeat ROP, this thesis will discuss ROP in detail in Chapter 3.

Chapter 3

Return-Oriented Programming

Return-oriented programming (ROP) is an improvement over the return-to-libc attacks described in Section 2.2.1. Instead of using whole functions existing in the standard C library, ROP utilizes code sequences existing in libc or other loaded libraries as building blocks to construct a gadget for the attack. Because the instruction set of the Intel x86 architecture is very dense and can also be interpreted in multiple ways (as there is no requirement for an instruction to be aligned in memory), it is simple to find and construct code sequences that end with a `ret` instruction in large libraries. Such sequences are called "gadgets." Because it is possible to find many code sequences that can be used for gadget construction, Shacham [17] shows that it is possible to construct a Turing-complete machine from such gadgets, allowing an attacker to execute any arithmetic and logic operations and to execute any function with the same privilege as the program.

Because of its power, ROP gained considerable interest from the computer security community. This has led to the development of many techniques that attempt to defend against ROP and the whole class of code reuse attack. Since there are many attacks and circumvention techniques related to ROP, this chapter will discuss variations of the ROP attack first, then talk about defenses and their effectiveness.

3.1 Variants of ROP Attacks

3.1.1 ROP Using `pop` and `jmp` Instructions

In the original ROP paper by Shacham [17], all building blocks that are used to construct an attack gadget need to end with a `ret` instruction. As such, there have been many attempts to use patterns of execution of `ret` instructions to defeat ROP. However, Checkoway et al. [18] show that it is possible to make an attack gadget that does not result in `ret` instruction being executed at all. For example, `pop` and `jmp` instructions on Intel x86 can be used as a replacement to achieve the same result as the original ROP attack. The replacement can also be constructed to have the same Turing-completeness property.

3.1.2 Jump-Oriented Programming

In addition to using `pop` and `jmp` instructions, Bletsch et al. [15] demonstrate that only `jmp` instructions are required for the attack. Instead of finding code sequences that end with `ret` or `ret`-like (`pop` and `jmp`) instruction, they show that by chaining "functional gadgets," which are sequences of instructions that contain `jmp` instructions, this attack can accomplish the same goal as ROP. The attack can also circumvent many anti-ROP defenses that monitor the execution of `ret` instructions in short sequence.

3.2 Detection-Based Defenses

3.2.1 ROPdefender

ROPdefender attempts to defeat ROP by preventing return addresses on the call stack from being exploited by ROP. It watches for all `call` and `ret` instructions. For each `call` instruction, it puts an expected return address into the shadow stack, a stack that is hidden from the application. When a `ret` instruction is called, ROPdefender pops a

return address from the shadow stack and checks whether it matches the address that the `ret` instruction is going to return. If there is a mismatch, ROPdefender terminates the program. Since ROPdefender is an instrumentation-based tool, it does not require any modification to binaries. However, the average run-time overhead is around 200%, which is high, even when compared to other detection-based defense approaches [19].

3.2.2 kBouncer

kBouncer is an implementation of two ROP defense approaches. First, kBouncer checks that for each execution of a `ret` instruction, the return address that appears on the stack points back to valid call sites, which are locations that occur immediately after `call` instructions. Second, kBouncer checks a gadget-chain length, which is the number of instructions executed in each taken indirect branch. It uses the Last Branch Record (LBR), a feature in recent Intel CPUs, to record information about indirect branches. By running these checks only during each system call, kBouncer has an overhead of only 4%. However, since kBouncer is implemented using the Microsoft EMET toolkit, kBouncer only works on Windows [20].

3.2.3 ROPecker

ROPecker attempts to prevent ROP attacks by looking at the gadget-chain length, similar to kBouncer. However, the ROPecker system is built on the Linux kernel level, instead of using the EMET toolkit, which requires binary rewriting. ROPecker also uses a sliding window mechanism and offline analysis to reduce overhead and provides more accuracy in indirect branching detection [21].

3.2.4 Circumvention

While several detection-based defenses claims to retain compatibility with legacy software, and while some of them are very efficient, these defenses are not effective at preventing attackers who know the implementation details of such protections.

For ROPdefender, attack gadgets that avoid using a `ret` instruction can escape the defense easily.

kBouncer also has similar flaws. First, if an attacker uses only called-preceded gadgets, the call-preceded policy can be circumvented. Carlini and Wagner [22] show that only 70KB of binary code is enough to construct such gadgets. Second, the gadget classification mechanism, which uses the gadget-chain length to detect ROP attacks, can be circumvented by issuing multiple "long gadgets." These are gadgets long enough for kBouncer to not classify them as ROP attempts. With enough long gadgets, Last Branch Record (LBR) would not be able to store all branching information and the older information will be flushed. As such, real ROP attack gadgets would still be able to execute because they are hidden from detection by kBouncer .

ROPecker also has similar issues. However, since ROPecker sacrifices performance to run ROP detection more frequently, overloading the LBR to flush traces of ROP attacks does not always work with ROPecker. However, by using the same approach repeatedly, Carlini and Wagner [22] show that there must be a period that ROPecker can not detect the attack. A technical report by Schuster et al. [23] also mentioned similar circumvention techniques for kBouncer and ROPecker.

3.3 Address Space Layout Randomization (ASLR)

Address Space Layout Randomization (ASLR) is another technique that was created to defend against both code-injection and code-reuse attacks. To prevent the attacks, before running a program, a system with ASLR randomly assigns address offsets to the base of each segment in the program's binary, instead of just using the offsets provided in the binary as in the system without ASLR. The randomness makes it more difficult for an attacker to perform a code-reuse attack, as the attacker usually needs to be able to determine the address of gadgets or functions to prepare the attack [24].

Similar to DEP, ASLR is widely adopted and supported by major operating systems [9]–[11]. However, to support ASLR, a legacy program needs to be recompiled with the position-independent executable (PIE) features enabled.

3.3.1 Variants of ASLR

3.3.1.1 Address Space Layout Permutation (ASLP)

Address Space Layout Permutation (ASLP) provides a way for a legacy program compiled without PIE to be able to use address randomization without the need for recompiling the program. ASLP implements two approaches to enable this: user-level randomization and kernel-level randomization. For the user-level randomization, ASLP uses a binary rewriting scheme. For a user to enable the user-level ASLP for a program, the user runs the provided tool that relocates binary regions to different memory addresses. The tool also detects all references to such regions and modifies them so that the program can still work as expected [25].

On the other hand, the kernel-level implementation of ASLP allows users to run all programs without any modification, as the similar permutation scheme is done transparently in the kernel [25].

3.3.1.2 Instruction Location Randomization (ILR)

Instruction Location Randomization (ILR) improves ASLR by randomly placing every instruction in memory. For a program to run with ILR, the program needs to undergo offline analysis using a disassembler and branch and call site analyzers. After that, the program can run on a per-process virtual machine. The virtual machine is required as all instructions appear randomly in the memory. Therefore, without the virtual machine, a program can not be properly executed. However, because of this, it is much more difficult to perform ROP attack on a system with ILR than one with ASLR [26].

3.3.1.3 ASLR-Guard

ASLR-Guard provides protection to code pointers in two ways. First, ASLR-Guard separates memory between code and data regions. In particular, ASLR-Guards provides a way for "sensitive stack data", including return addresses, to be stored separately from other data in the call stack. Second, ASLR-Guard encrypts code

pointers using XOR encryption so that an attacker cannot read the actual memory address to which the code pointers point. Because of the encryption, ASLR-Guard also prevents attackers from overwriting code pointers as the attackers would not be able to encrypt a location without a correct key [27].

3.3.1.4 Timely Address Space Randomization (TASR)

Because ASLR only randomizes offsets of memory regions when programs start, if an attacker can find a way to read memory, the attacker might be able to learn the offsets. For example, derandomization attack, shown in Section 3.3.2.1, allows an attacker to guess the offsets almost immediately. Timely Address Space Randomization (TASR) mitigates the problem by re-randomizing locations of executable regions and updating every code pointer that points to such locations between each system call. With this protection, the attacker would not be able to learn the correct memory address to create an attack gadget on time [28].

3.3.2 Circumvention

3.3.2.1 Derandomization Attack

Derandomization attack is a technique to attack ASLR on 32-bit machines by reducing the search space of offsets. By using addresses placed on the stack by the program itself, derandomization attacks allows an attacker to reduce the complexity of searching for the right offset of the libc segment from 25 bits to 16 bits. With only 16 bits of search space, attackers can use a brute force approach to find the right offset to construct an attack. According to Shacham et al., re-randomizing frequently would not help prevent the attack, as it can add no more than one bit of entropy [29].

On the other hand, 64-bit machines are not affected by this attack because an attacker would need to use brute force in 40 bits of address space, which is not feasible in most cases [29].

3.3.2.2 Just-In-Time Code Reuse

Just-In-Time Code Reuse is a technique that enables attacks on an application with multiple memory disclosure vulnerabilities, even with ASLR enabled. With multiple memory disclosures, ASLR is undermined as the attacker would simply be able to search through all mapped memory. In some cases, an attacker would be able to construct gadgets on-the-fly and only need to search through only a small address space. This attack works on both 32-bit and 64-bit machines and is also platform-independent [30].

3.3.2.3 Blind ROP

While there are many code-reuse attack approaches that circumvent ASLR and other protections, those attacks are difficult to perform on many targets that use proprietary and closed-source software. Even with the source code, if a binary is compiled differently on each specific machine, it is still difficult to launch a ROP attack as the attacker would not know the specific location of the desired gadgets.

Blind ROP (BROP) shows that it is possible to attack the targets without access to their binary, even with ASLR, DEP, and stack canaries enabled. By using generalized stack reading technique to find ROP gadgets, BROP can find locations of libc functions that appear on the Procedure Linking Table (PLT), allowing the attacker to construct a ROP gadget using such information. [31]

3.3.2.4 Side Channel Attacks

Because just-in-time code reuse and blind ROP attacks require code to be read from memory, some variants of ASLR make it more difficult for an attacker to do so, including ILR in Section 3.3.1.2. However, Seibert et al. show that the attacker can use side channel attacks to leak information indirectly, without having to read code from memory. For example, if an attacker can overwrite a variable in a loop, the attacker might be able to read information by repeatedly recording timings that the application takes before responding back to the attacker and using statistical analysis

to determine information from the data. The paper also demonstrates that this kind of attack works even in the situation where the attack is performed remotely between two machines in the same network [32].

3.4 Annotated Language

The C language allows developers to perform a number of actions that cannot be done directly in higher level languages, including accessing memory without any data type checking, dereferencing pointers without any bound check, and using native assembly. Such freedom helps developers to write code that interacts directly with hardware, allowing them to write very high-performance software. However, those features also have an unintended consequence that makes it much more dangerous for developers to accidentally introduce bugs, especially security bugs, to the software. Annotated language is a solution that has been developed to provide stronger memory safety and reduce software bugs by limiting what developers can do with C language.

3.4.1 Cyclone

Cyclone is one of the earliest attempts to modify the C language. Cyclone adds multiple restrictions on what developers can do, including restricting pointer arithmetic and disallowing any use of `setjmp` and `longjmp`. It also enforces run-time bound checking on every pointer with the "fat pointer" scheme. The benchmark shows that Cyclone has the average runtime overhead of 16% more than the original programs in C [33].

3.4.2 CCured

CCured also imposes restrictions on pointer usage. However, CCured gives more flexibility to developers to make their decisions to trade off safety with performance. CCured also uses type interference to reduce the overhead of the program by making static verification and removing unnecessary checks that prove to be impossible. CCured has an overhead of 3% more than the same code compiled in C without any

protection [34].

3.4.3 Rust

Rust is a programming language that was created with the intention to make a program more secure. However, the syntax, while similar to C/C++, is not designed to be compatible with the C/C++ code. Rust also supports the purely functional language paradigm, which is not originally available in C++ [35], [36].

3.5 Bound-Based Defenses

In light of the ineffectiveness of DEP, stack canaries, and ASLR in preventing code-reuse attacks, techniques to restrict a range of addresses that pointers can point to are developed with the goal of guarantee complete spatial memory safety. This section will discuss three implementations of such approaches: HardBound, SoftBound and Baggy Bound.

3.5.1 HardBound

HardBound protects a pointer from overflowing its intended range by transparently storing information about the base and bound, ranges of addresses that the pointer can point at. By doing so, HardBound provides spatial memory safety, preventing overflow from happening in the first place. In addition to that, Hardbound uses compression to store the base and bound of each pointer. This allows HardBound to reduce memory access as more pointer information would be able to fit in the L2 cache. HardBound also adds a "tag metadata space" to store information on whether each memory address is a pointer data type so that the machine is not required to access the base and bound information for non-pointer data, improving the overall performance [37].

HardBound does not guarantee temporal memory safety and requires modification to the hardware. However, it provides very low overhead compared to the earlier fat

pointers scheme.

3.5.2 SoftBound

SoftBound uses a similar protection mechanism to that in HardBound, but without requiring any hardware modification. SoftBound is implemented by inserting a check before every pointer dereferencing and manipulation. By doing a transformation of at the compile time, SoftBound does not require any source code change. However, it is still necessary to recompile a program to utilize SoftBound. The average overhead that SoftBound incurs is also much higher than HardBound does, with 67% compared to 10% [38].

3.5.3 Baggy Bound

Baggy Bound uses a technique to similar SoftBound's. However, instead of adding two additional pointers to represent the base and bound, Baggy Bound uses spare bits that pointers in 64-bit architecture already have to store the base and bound. Baggy Bound stores the value of $\log_2(size)$ instead of the whole base and bound in each pointer, reducing numbers of bits needed and allowing faster check using bitwise operations. However, Baggy Bound requires that the size of all allocated memories be a power of two. Baggy Bound has an average overhead of only 12% and does not require any hardware or source code modification [39].

3.6 Code-Pointer Integrity (CPI)

Code-Pointer Integrity (CPI) is a technique to protect code pointers by storing them in a "safe region." The safe region is a region of memory that can be accessed only by modified instructions that have been statically analyzed by CPI, preventing attackers from overwriting code pointers. To implement the safe region, CPI uses a different method for each CPU architecture. For x86-32, CPI uses the segment protection mechanism in hardware. However, in x86-64 and ARM, CPI uses a technique similar

to ASLR to hide the location of the safe region [40].

CPI claims to provide very low overhead, only 2.9% for C and 8.4% for C++. The paper also discusses another implementation, called Code-Pointer Separation (CPS), that has an overhead of approximately half that of CPI, but with fewer security guarantees [40].

3.6.1 Circumvention

According to Evans et al. [41], a version of CPI that is implemented for x86-64 and ARM utilizes the location hiding technique to provide storage for code pointers. This version of CPI is subject to the same problem that ASLR has. By using side-channel attacks, similar to the one that is described in Section 3.3.2.4, an attacker can locate the safe region and modify the code pointers.

However, Kuzentsov et al. [42], the developers of CPI, argued that the attack as mentioned is only a flaw in the implementation, not in the CPI technique, and can be fixed by a small number of changes.

3.7 Control-Flow Integrity (CFI)

Control-Flow Integrity (CFI) prevents code-reuse attacks by constraining all control transfer instructions, including `ret` and `jmp`, and blocking the execution of these instructions that follows an unexpected execution path. CFI constructs a control-flow graph (CFG) by analysis. CFG is used to make a decision about whether an execution of instruction should be allowed. The CFG analysis can be performed with static analysis or execution profiling. The CFI enforcement can be done in multiple ways, including instrumentation of a binary [43].

3.7.1 Variants of CFI

3.7.1.1 Compact Control Flow Integrity and Randomization (CCFIR)

Compact Control Flow Integrity and Randomization (CCFIR) improves upon CFI by locating all indirect control-transfer instructions in a section on memory called the "springboard section." By doing so, CCFIR simplifies how CFI is validated for each instruction, resulting in the improvement of overall performance. CCFIR has an average overhead of 3.6% [44].

3.7.1.2 Control Flow and Code Integrity (CFCI)

Control Flow and Code Integrity (CFCI) allows an existing binary to be protected by CFI. CFCI implements a secure library loading model for secure loading, which is used to protect other types of attacks. CFCI has an average overhead of 14.37% [45], [46].

3.7.1.3 Cryptographically-Enforced Control Flow Integrity (CCFI)

Cryptographically-Enforced Control Flow Integrity (CCFI) uses message authentication codes (MACs) to enforce CFI at runtime. As such, CCFI can detect some usages of pointers that static analysis is not able to detect. CCFI takes advantage of AES-NI, a set of AES-related instructions in new Intel CPU models, to speed up cryptographic processes that are used to validate the MACs [47].

3.7.1.4 Opaque Control-Flow Integrity (O-CFI)

Opaque Control-Flow Integrity (O-CFI) uses both fine-grained code randomization and coarse-grained CFI to mitigate implementation-disclosure attacks, which are attacks that use information about program memory to construct a specific attack for each scenario. To hide control flow graphs from an attacker, O-CFI creates a bound lookup table at a random location in memory, preventing an attacker from reading information in the control flow graph [48].

3.7.2 Circumvention

3.7.2.1 Coarse-Grained CFI

Davi et al. show that while CFI is likely to be effective in preventing ROP attacks, many implementations of CFI use a coarse-grained approach to improve the average performance of programs. These coarse-grained solutions are insufficient to prevent attackers as Turing-complete ROP gadgets can still be constructed in such environments. Even the combinations of all coarse-grained defenses are not sufficient to prevent such attack [49].

3.7.2.2 Problem with Static Analysis

Similar to coarse-grained CFI, Goktas et al. demonstrate that CFI that utilizes only static analysis to create a control-flow graph is breakable [50].

3.7.2.3 Counterfeit Object-Oriented Programming (COOP)

Counterfeit Object-Oriented Programming (COOP) is an attack that focuses specifically on C++. COOP uses a chain of C++ virtual functions that exists in some C++ applications to construct a ROP gadget. Because CFI needs to allow legitimate flow that uses the chain, the attack is not detected by CFI implementations [51].

3.7.2.4 Control Jujutsu

Control Jujutsu is an attack that is based on the incompleteness of the control-flow graph creation. Because it is impractical to create a perfectly accurate control flow graph in a large-scale program, a control flow graph normally includes some edges of flow that are not possible in a legitimate execution of the program. Evans et al. show that even with the fine-grained CFI and strict control flow graph creation, an attacker might be able to utilize inaccuracy characteristic of the control flow graph to construct a ROP attack. They also show that such attacks are not only possible in theory, but also possible in many real-world applications, including Apache and Nginx. [52]

Chapter 4

Tagged Architecture

Tagged architecture is a type of computer architecture that stores a tag for each word in the memory. The tag can be used for multiple purposes. For example, a system can use the tag to store data type information for every variable in memory. The tag can also be used for debugging purposes or marking areas of memory for garbage collection. The idea of tagged architecture dates back as early as 1973 [53]. However, the majority of modern computer architectures, including the Intel x86 and ARM, still do not support the implementation of tagged architecture at the hardware level.

In this chapter, we will discuss two tagged architectures that have been developed with the goal of improving security of software: Capability Hardware Enhanced RISC Instructions (CHERI) and Programmable Unit for Metadata Processing (PUMP).

4.1 Capability Hardware Enhanced RISC Instructions (CHERI)

Capability Hardware Enhanced RISC Instructions (CHERI) is a modification of the 64-bit MIPS IV instruction set to support capability models, allowing the enforcement of memory policies and fault isolation at the hardware level. Similar to HardBound, CHERI provides complete memory safety. CHERI also supports an enforcement of capability, allowing a program to enforce specific protection in some sensitive memory

regions [54].

In CHERI, a 256-bit memory capability is used to enforce the protection. The capability contains the base and bound (64 bits each for the total of 128 bits) and permission fields (31 bits). The 97 unallocated bits can be used by the user-level application to store data. To prevent an attack from modifying the capability by overwriting it, one bit of tag is used for every 32 bytes of the memory to distinguish between capability and general-purpose data. CHERI uses the physical memory address instead of the virtual address in the tagging scheme to avoid the need for memory address translation [54].

The implementation of CHERI is based on Bluespec Extensible RISC Implementation (BERI). However, by adding the tagged architecture and capability supports, CHERI uses 32% more logic elements than BERI and has an overhead of 8.1%, compared with BERI [54].

In an attempt to retain the compatibility with C programs, Chisnall et al. [55] analyzed 13 open-source applications to observe the specific usage of C features and idioms that might break when compiling the applications for the CHERI architecture. They modified CHERI instructions and created an abstract machine that allowed the modified CHERI to run these C programs with little to no modification to the source code.

4.2 Programmable Unit for Metadata Processing (PUMP)

Programmable Unit for Metadata Processing (PUMP) is an architectural model that generalizes tagged architecture. PUMP allows programming of the processor, enabling a user-level program to enforce customized tag propagation and memory enforcement policies that are suitable for each environment. Specifically, PUMP allows a program to enforce policies that use an instruction and the values of tags as an input. With the input, the policy can be created to propagate new tag values into the output or to

block the execution of an instruction [4].

The flexibility of this policy customization makes it possible to implement many forms of memory enforcement, including spatial and temporal memory safety and control-flow integrity. PUMP can also be used for trait tracking without any hardware change. PUMP has a small overhead of approximately 10% of running time and under 60% for memory usage [4].

4.2.1 Policies

PUMP can enforce a wide range of policies, including memory protection, capabilities, data type, taint tracking, and invariant checking. A program can enforce a policy by either inserting the policy when the program is loaded or creating a miss handler that inserts a policy on-demand. A miss handler is called every time the policy is not found in the rule cache [4].

Each policy can contain filters for the following inputs: program counter, current instruction tag, and tag on both input and output arguments of the instruction. Each policy can take multiple actions, including blocking the instruction from being executed, modifying tags of the result value, and setting a program counter [4].

4.2.2 Implementation

To implement PUMP, a processor is modified to include a PUMP rule cache. The rule cache contains logic gates that receive execution information from components in the original part of the processor. The rule cache also contains a small associative memory that is used for the caching mechanism. The rule cache produces three outputs for each instruction. These outputs are program counter, result tag, and binary value indicating whether a matched policy is found in the cache memory.

PUMP uses several techniques to optimize for high performance, including tag compression and miss-handler acceleration. PUMP also uses an additional cache for tag translation to reduce the energy usage.

4.2.3 Policy Correctness

By using dynamic sealing, a proof mechanism that is used to prove the correctness of several cryptographic algorithms, an abstract and symbolic machine can be constructed to prove micro policies in PUMP. These constructions are shown to be able to prove multiple micropolicies, including ones for memory safety, control-flow integrity, and memory compartmentalization [56].

Chapter 5

RISC-V

The content of this chapter is based on the RISC-V instruction set manual volume I [57] and volume II [58].

RISC-V is an open-source instruction set specification developed by the Computer Science Division at UC Berkeley with the goal of becoming a standardized instruction set for both industrial and academic usages. RISC-V is designed to be simple and extensible.

In this chapter, we will provide an overview of the design of RISC-V. We will also discuss Spike, an instruction simulator for RISC-V, and its components.

5.1 Instruction Set

Instructions in RISC-V ISA are of variable length, with a minimum length of 32 bits, and can be extended to any size depending on support from the hardware. In this section, we will focus on the instructions in the general-purpose ISA, as defined in the RISC-V instruction set manual.

The general-purpose instructions are divided into two categories: base instruction set and extension instruction set. Instructions in the base instruction set are required to be supported by all RISC-V-compatible CPUs. However, a hardware designer can choose to exclude instructions in the extension instruction set. Numbers of instructions

in the base instruction set and each extension are shown in Table 5.1.

Extension	**32 bits**	**64 bits**	**Total**
Base Integer	47	15	62
Standard Extension for Atomic (A)	11	11	22
Standard Extension for Multiplication and Division (M)	8	5	13
Standard Extension for Single-Precision Floating Point (F)	34	4	38
Standard Extension for Double-Precision Floating Point (D)	26	6	32
Total	126	41	167

Table 5.1: Numbers of general-purpose RISC-V instructions, categorized by extension.

Instructions in RISC-V ISA can also be classified by its "opcode", which is the seven lowest bits in each instruction. For all instructions with a length of greater than or equal to 32 bits, the lowest two bits of instructions need to have values of 11 in binary. Table 5.2 shows a list of opcodes in RISC-V.

Opcode (in binary)	**Name**	**Number of Instructions**
0000011	LOAD	7
0000111	LOADFP	2
0001111	MISCMEM	2
0010011	OPIMM	12
0010111	AUIPC	1
0011011	OPIMM32	4
0100011	STORE	4
0100111	STOREFP	2
0101111	AMO	22
0110011	OP	18
0110111	LUI	1
0111011	OP32	10
1000011	MADD	2
1000111	MSUB	2
1001011	NMSUB	2
1001111	NMADD	2
1010011	OPFP	50
1100011	BRANCH	6
1100111	JALR	1
1101111	JAL	1
1110011	SYSTEM	16

Table 5.2: List of RISC-V opcodes used by general-purpose instructions.

5.2 Registers

RISC-V is designed to have 31 general-purpose integer registers, `x1-x31`, and 32 privileged control registers (PCRs), `pcr0-pcr31`. (The `x0` register is hard-wired to always contain zero.) Hardware implementations of RISC-V should treat all general-purpose registers in the same way. However, the RISC-V port of the Linux kernel has a convention for register usage. For example, Register `x1` is called `ra` and should be used to store a return address [59].

In contrast to the general-purpose registers, each PCR has a specific function at the hardware level. For example, `pcr0` is a status register that contains values of the instruction pointer and other hardware-level settings.

5.3 Spike

Spike is a hardware simulator for RISC-V based on QEMU, an open-source hypervisor. Spike emulates each instruction directly, which makes it slower than some modern hypervisors that utilize hardware acceleration to accelerate the simulation. However, because of the simplicity of the emulation, Spike is suitable as a tool for experimenting with changes to the RISC-V instruction set [60].

5.4 Proxy Kernel (PK)

Since Linux is a full-feature kernel, it contains many components that are not required for experimentation and boots slowly in a system with high overhead, including in the Spike simulator. Proxy Kernel is a kernel implementation that allows a program to run on Spike without having to use the Linux kernel. It acts as a minimal layer to support applications that are compiled with Newlib, a C standard library for embedded systems [61].

Chapter 6

TAXI

This chapter is a summary of TAXI, an approach to defeat code-reuse attacks with minimal hardware modification. The content of this chapter is based on the theses by Issac Evans [1], Sam Fingeret [2], and Julián González [3].

TAXI is a set of modifications to the RISC-V instruction set with the intention of preventing code reuse attacks with tagged architecture. It aims to preserve compatibility with C applications at the source code level. In this chapter, we will discuss TAXI and its implementation.

6.1 Tagged Architecture

In TAXI, tagged architecture is implemented by dividing memory into two sections: data memory and shadow memory. For every 64 bits of data, there is an 8-bit tag that is associated with the data. The tag is contained inside of the shadow memory, hidden from the normal operation of user-level applications. The tag is used to store metadata. Specifically, the tag can be used to determine whether the data is a return pointer or a function pointer. The tag can also contain other information as needed, depending on each protection scheme.

6.1.1 Tag Unit

To enable enforcement of the tagging scheme at the hardware level, a "dedicated hardware tag unit" has been added to P-TAXI. For every instruction executed, information about the execution is provided to the tag unit. This information includes the instruction itself, the tag values of parameters provided to the instruction, and the value of the program counter. The tag unit processes these inputs in parallel to the actual execution of the instruction. By considering the policies that are hard coded in the tag unit, the tag unit makes a decision about whether the instruction should be allowed to execute. In the case that a policy is violated, the tag unit will trigger a trap signal. The signal then propagates to the operating system to take further action.

6.1.2 Tag Cache

Because there is a tag attached to every word in the memory, without any caching, tag values will need to be retrieved from the memory directly for every instruction execution. This would result in incurring tremendous overhead. TAXI solves this problem by adding tags to every level of caching. In particular, tag registers are added to accompany every register. For the L1 and L2 caches, there are also tag caches added for every word in the cache with the same level of performance. In addition, between the L2 cache and the memory, a large tag cache is added to reduce a number of direct access to the memory.

6.2 Policies

TAXI provides three sets of policies that can be enforced at the hardware level: return address protection, linearity of return address, and data blacklisting.

6.2.1 Return Address Protection

With tagged architecture, TAXI prevents an attacker from modifying return pointers in the stack by tagging every return address. For every function call, TAXI tags a

return address that returned from an execution of the `jal` instruction, a jump-and-link instruction used for function calls in RISC-V. When the return address is saved or restored from the memory, the tag is also propagated in the same flow. When a function returns, TAXI checks for the existence of the tag in the return address stored in the register that contains the return address. If the tag does not exist, TAXI will trigger a trap.

6.2.2 Linearity of Return Address

To ensure that an attacker cannot create an attack gadget that copies return addresses with valid tags and uses them afterward to perform a code-reuse attack, TAXI also provides a set of policies called "blacklist no partial copy." With this policy, TAXI ensures that there is only one copy of each valid return address existing at a time for each return address created by calling a function. To achieve this goal, for every move of the return address, including ones between registers and to/from the memory, TAXI clears a tag existing in the source before adding a tag to the destination.

6.2.3 Data Blacklisting

Without support from compilers, it is difficult for both the CPU and the operating system to determine types of data in each memory word. However, in the RISC-V architecture, there are multiple commands that can be used to load and store data to/from the memory, allowing a program to load values of various sizes, ranging from 8 bits to 64 bits. Since TAXI is implemented on the 64-bit RISC-V architecture, we know that all pointers need to have the size of 64 bits. Because of this, for all non-64-bit load and store operations, we can be certain that both the source and the destination of these operations are not pointers. As such, TAXI marks these registers and memory locations as data only, preventing the dereferencing of certain non-pointer values.

6.3 Performance

With the tag cache mechanism described in Section 6.1.2, TAXI has very low overhead when the size of the tag cache is sufficiently large. With 8MB of tag cache, the overhead of TAXI is lower than 5%, on average, when benchmarking using the SPEC2006 test suite.

Chapter 7

P-TAXI

In this thesis, we made several modifications to TAXI, described in Chapter 6, allowing an enforcement of TAXI to be programmable by user-level programs. We called our modifications P-TAXI. This chapter will discuss our contribution and the design of P-TAXI.

7.1 Contributions

Inspired by tagged architecture and PUMP described in Chapter 4, we modified TAXI to allow it to be programmable in a way that is similar to PUMP. However, P-TAXI still retains the size of a tag for each word in the memory to 8 bits. Specifically, this thesis has made the following contributions:

- Modified TAXI to allow each application to have a distinct set of memory safety enforcement policies. The set of policies can be chosen by either a developer of the program or a user who runs the program.
- Created a policy specification and framework that are flexible and universal, allowing application developers to implement policies for various purposes.
- Developed a policy enforcement simulator as an add-on to TAXI such that we can test the correctness of policies and measure their performance.

- Demonstrated that with these modifications, P-TAXI can be used in multiple scenarios for multiple purposes, including but not limited to defeating code-reuse attack, debugging, and trait tracking.

7.2 Threat Model

Some defense approaches assume that there might be a malicious code existing and running on a system. However, such threat models are not reasonable for P-TAXI because the implementation is created to allow developers to have full control of the enforcement policy. Therefore, for P-TAXI, we make an assumption that a program running on the system might contain some vulnerabilities that allow an attacker to perform buffer overflow attacks. However, the program is not intentionally developed to be malicious.

7.3 Design

Because P-TAXI is not created to solely address a specific class of security bugs, we designed P-TAXI to allow developers to add a set of policies to their program. To make this possible, we added an instruction called `TAGPOLICY` that receives policies from a program and stores the policies in a separate memory called "policy storage." Since the size of policies is marginal, in hardware implementation, the policy storage is intended to be a component inside the CPU, similar to the L2 cache.

After the policies are added to the policy storage, an application can enable the programmable policy enforcement unit by running the `TAGCMD` instruction. With the enforcement enabled, for each instruction executed, the enforcement unit will scan through a set of policies added specifically for the application and determine whether the policy matches. If there is a policy that matches with the instruction, the action specified in the policy will be performed.

7.3.1 Policy

Each policy contains two components: filter and action. A filter is a set of conditions that is used to specify whether the policy should be applied in a specific scenario. An action is an outcome that should occur if the policy matches with all filters.

7.3.1.1 Filter

P-TAXI supports multiple filter conditions as shown in Table 7.1.

Type	Name	Description	Bits
Filter	TAG_ARG1	Mask and match bit field for tag on ARG1.	16
	TAG_ARG2	Mask and match bit field for tag on ARG2.	16
	TAG_OUT	Mask and match bit field for tag on OUT.	16
	RS1	Mask and match bit field for RS1.	10
	RS2	Mask and match bit field for RS2.	10
	RS1VAL	Mask and match bit field for the value of RS1.	16
	RS2VAL	Mask and match bit field for the value of RS2.	16
	PRIV	Mask and match bit field for current privilege state.	16
	INSN_TYPE	Instruction type filter.	8
	IGNORE_COUNT	Numbers of policy matches to skip before starting to take any action.	8
Action	ACTION	Action to do.	8
	TAG_OUT_SET	Bit field of value to set to tag	8
	TAG_OUT_TOMODIFY	Bit field to set which bits of tag value to be modified.	8
	PRIV_SET	Bit field of value to set to privilege state.	8
	PRIV_TOMODIFY	Bit field to set which bits of privilege state value to be modified.	8
Total			172

Table 7.1: List of fields available for each P-TAXI policy.

7.3.1.2 RISC-V Instruction Classification

Some RISC-V instructions can be used in multiple ways. For example, the `ADDI` instruction is used for both adding and copying values. The `JALR` instruction is also used for both jump and return. These usages of instructions make it complicated to implement a policy based solely on the names of instructions. P-TAXI solves this problem by reclassifying RISC-V instructions into multiple instruction types, shown in Table 7.2. As shown in the table, P-TAXI also separates load and store instructions into two groups, based on whether the instructions are intended for 64-bit data. The purpose of this is to allow application developers to create a policy that acts differently when loading and storing data that can be interpreted as a pointer.

7.3.1.3 Action

P-TAXI supports actions as shown in Table 7.3.

7.3.2 Commands

P-TAXI has two additional commands that can be called by user-level programs: `TAGCMD` and `TAGPOLICY`.

7.3.2.1 `TAGCMD`

The `TAGENFORCE` instruction in TAXI is renamed to `TAGCMD` to represent the purpose of this instruction in P-TAXI. In TAXI, `TAGENFORCE` can be used only to enable or disable tag enforcement. However, in P-TAXI, we modified this instruction to also allow user-defined behaviors by policies.

`TAGCMD` accepts three arguments: command code, an input register, and an output register. A policy can be added to P-TAXI to make the execution of `TAGCMD` with the specified command code behave in the way that is defined in the policy. For example, in normal conditions, `TAGCMD` will copy a value from the input register to the output register. A policy with the `GETTAG` action can make `TAGCMD` acts as a command to retrieve tag values from registers.

P-TAXI Type	Description	RISC-V instructions	Arguments		
			ARG1	ARG2	OUT
LOAD	Instructions that load a value from memory for non-64-bit data.	LB, LH, LW, ...	Memory	N/A	Register RD
LOAD64	Instructions that load a value from memory for 64-bit data.	LD	Memory	N/A	Register RD
STORE	Instructions that store a value to memory for non-64-bit data.	SB, SH, SW	Register RS2	N/A	Memory
STORE64	Instructions that store a value to memory for 64-bit data.	SD	Register RS2	N/A	Memory
COPY	ADDI instruction that is used to copy values between registers	ADDI (imm=0)	Register RS1	N/A	Register RD
OP	Operation instructions.	ADD, SUB, XOR, OR, AND, ...	Register RS1	Register RS2	Register RD
OPIMM	Register-immediate instructions.	ADDI (imm≠0), SLTI, XORI, ...	Register RS1	N/A	Register RD
JAL	Jump-and-link (JAL) instruction.	JAL	Target Address	N/A	Register RD
JALR	Non-return Jump-and-link Register (JALR) instruction.	JALR (non-return)	Register RS1	Target Address	Register RD
RETURN	Return JALR instruction.	JALR (return)	Register RS1	Target Address	Register RD
TAGCMD	TAGCMD instruction.	TAGCMD	N/A	N/A	N/A
TAGPOLICY	TAGPOLICY instruction.	TAGPOLICY	Register RS1	Register RS2	N/A
SETTAG	SETTAG instruction.	SETTAG	Register RS1	Register RS2	N/A
UNKNOWN	Other instructions not matched with any types.		Register RS1	Register RS2	Register RD

Table 7.2: List of P-TAXI instruction types.

Action	Description
CONTINUE	Continue to apply next policies.
ALLOW	Allow the execution of the instruction and stop the policy processing for this instruction.
BLOCK	Block the execution of the instruction (cause trap) and stop the policy processing for this instruction.
GC	Do stack garbage collection and stop the policy processing for this instruction.
DEBUG_LINE	Show brief debug information.
DEBUG_DETAIL	Show detailed debug information.
GETTAG	Make TAGCMD returns the tag value of RS2 register (should use only with TAGCMD instruction).

Table 7.3: List of P-TAXI actions.

Command code zero is reserved for enabling tag enforcement because there cannot be a rule to enable policy enforcement, as the processing of policies would not be enabled at the time the first `TAGCMD` instruction is called.

Because `TAGCMD` can be used to both get and set tag values, we decided not to implement separate commands to get and set tags. However, an implementation of get and set tags is included in the user-level library of P-TAXI.

7.3.2.2 `TAGPOLICY`

The `TAGPOLICY` instruction is created to allow a program to add policies to the policy storage. `TAGPOLICY` accepts three 64-bit integer registers, representing a policy. The three integers are generated from serialization of "policy struct". Because of this, each policy can contain values of at most 192 bits. However, this can be modified easily if there is a situation that requires an extension to the size limitation.

7.3.3 Privilege Bits

To allow state-aware policies in P-TAXI, 8 privilege bits are allocated for each user-level program. These privilege bits can be used in multiple ways, including to implement

memory compartmentalization. For example, when a program enters a segment that need to interact with sensitive data, the program can enable a privilege bit so that a different set of policies is used in the execution. A policy can be created to take actions only when the privilege bits are in a specific state. The policy can also modify the privilege bits as needed.

7.4 Implementation

This section described how major components of P-TAXI are implemented. However, the source code of the P-TAXI simulator for Spike can be found in Appendix A.

7.4.1 Application-Specific Policies

P-TAXI uses a privilege control register (PCR) to allow each application to have a distinct set of policies. When a policy is added for the first time in the life cycle of an application, the value of the `status` PCR will be modified to include a context ID, an identifier that is used to identify which set of policies to enforce for the application. P-TAXI uses the `status` PCR for this purpose because it is expected to be saved and restored every time an operating system engages in context-switching between two processes. However, this limits the number of policy-enabled processes that can be run at the same time to 127. If more than 127 processes are required, it is possible to modify P-TAXI to use a separate PCR. However, the kernel of an operating system would need to be modified to save and restore the PCR while making a context switch.

7.4.2 Policy Detection

As shown in Section 7.3.1, each policy in P-TAXI contains multiple conditions. For efficiency, P-TAXI loads and stores tag values only when it needs to check or to update the values. P-TAXI also uses the same tag caching mechanism as developed in TAXI to avoid excessive access to the memory.

In Spike, policy detection is implemented to process each policy sequentially.

However, it is likely that policy detection can be done more effectively in hardware as a chip can be designed to process multiple policies in parallel.

7.4.3 Policy Enforcement

After a policy is determined to be matched, P-TAXI examines the policy and selects an appropriate action to take as specified by the policy. If the policy includes the `TAG_OUT_TOMODIFY` field with a non-zero value, the tag value at the location of the output of the instruction will be modified before P-TAXI takes any specified action.

To block the execution of an instruction, P-TAXI uses the same trap mechanism as in TAXI. P-TAXI also allows policies to enforce stack garbage collection. This is done by monitoring the stack pointer register and clearing the unallocated stack memory area when the `GC` action is requested.

However, since the `GC` action is likely to need more than one CPU cycle and the RISC-V architecture requires the execution of an instruction to use no more than one CPU cycle, in the actual hardware implementation, it might be more feasible to use a post-trap-handler approach. In the post-trap-handler approach, the CPU causes a trap to allow an operating system to perform the specified action instead of performing the action directly at the hardware level. This would allow more than one cycle of execution of complex actions.

7.4.4 User-Level Libraries

We created a set of libraries to allow a user-level programs to interact with P-TAXI components without having to write inline assembly code. To use these libraries, the program includes a header file that contains all basic sets of policies. The header file is called `ptaxi.h`. Listing 7.1 shows how a program can use P-TAXI in user-level code.

```c
#include <stdio.h>
#include "ptaxi.h"

#define TAGBIT 1

void __attribute__ ((constructor)) ptaxi_app_policy() {
  ptaxi_policy_return_address(TAGBIT);
  ptaxi_enforce_policy();
}

int main() {
  printf("Hello World\n");
    //...
  return 0;
}
```

Listing 7.1: Example C program that utilizes `ptaxi.h`.

7.4.5 `LD_PRELOAD` Environment Variable

`LD_PRELOAD` is a feature in Linux that allows dynamic loading of a library. By using `LD_PRELOAD`, it is possible to load any shared library into a program without having to recompile the program. To use this feature, a user sets the `LD_PRELOAD` environment variable by entering `export LD_PRELOAD=<path to a library>` in a terminal shell before starting the program. Every program that executes subsequently will have the specified library preloaded [62].

With P-TAXI, users can create a library to inject policies into a program. Listing 7.2 shows sample library code that can be compiled to run with `LD_PRELOAD`. This code can be compiled as a library with GCC.

```c
// To compile, run "riscv64-unknown-linux-gnu-gcc -fPIC -shared -o
   lib_gc.so test_gc.c"
#include "ptaxi.h"

void __attribute__ ((constructor)) ptaxi_inject_policy() {
    ptaxi_policy_gc();
    ptaxi_enforce_policy();
}
```

Listing 7.2: Example C library that can be loaded into existing programs to enable P-TAXI via `LD_PRELOAD`.

Chapter 8

Sets of Policies for P-TAXI

Because P-TAXI is programmable, there are unlimited numbers of sets of policies that can be implemented for user-level programs. In this chapter, we show basic sets of policies that are useful to apply to typical applications. Since the actual code to enforce these policies and their test cases is verbose, they are shown in Appendix B.

8.1 Base Sets of Policies

Because P-TAXI is designed to keep modification to hardware as marginal as possible, P-TAXI only provides two commands, `TAGCMD` and `TAGPOLICY`. Additional policies need to be added to P-TAXI by software to allow more complex policies. For example, to be able to set tags and privilege bits from a program, a program need to add a policy to change the behavior of the `TAGCMD` command. Several base sets of policies are provided in the P-TAXI user-level libraries. These base sets of policies can be used by both user-level programs and more advance sets of policies. These base sets of policies are described in Table 8.1. The implementation of these set of policies are shown in Table 8.2.

Name	Description
PROPAGATE	Propagate tags in the output argument when instructions in a specified type is called.
CLEAR	Clear tags in the output argument when instructions with a specified type is called.
SETTAG	Allow a user-level program to call TAGCMD to set tags with the registers.
CLEARTAG	Allow a user-level program to call TAGCMD to clear tags in the registers.
GETTAG	Allow a user-level program to call TAGCMD to retrive tags in the registers.
SETPRIV	Allow a user-level program to call TAGCMD to set privillage bits.
CLEARPRIV	Allow a user-level program to call TAGCMD to clear privillage bits.

Table 8.1: List of base sets of policies implemented in the P-TAXI user-level libraries.

Name	#	Filter		Action		
		Instruction Type	Additional conditions	Action	Output tag	Priv. Bits
PROPAGATE	1	<Specified>	TAG(ARG1) = 1	-	Clear	-
	2	—"—	TAG(ARG1) = 0	-	Set	-
CLEAR	1	—"—	N/A	-	Clear	-
SETTAG	1	TAGCMD	RS1 = <Specified>	-	Set	-
CLEARTAG	1	—"—	—"—	-	Clear	-
GETTAG	1	—"—	—"—	GETTAG	-	-
SETPRIV	1	—"—	—"—	-	-	Set
CLEARPRIV	1	—"—	—"—	-	-	Clear

Table 8.2: List of P-TAXI policies used to implement the base sets of policies.

8.2 Return Address Protection

Return address protection is implemented in TAXI to prevent an attacker from modifying return pointers in the call stack to initiate code-reuse attacks. However, TAXI implementation is accomplished by hard-coding the behavior of instructions. With P-TAXI, we developed a set of policies that behave in the same way as in TAXI, but without any change to the hardware. Table 8.3 shows the set of policies that provides return address protection.

#	**Filter**		**Action**	**Rationale**
	Instruction Type	**Additional conditions**		
1	JAL	N/A	Set TAG(RD) = 1	Set tags on return address values.
2	RETURN	TAG(ARG1) = 0	BLOCK	Block a return attempt without a valid tag.
3	STORE64, LOAD64, COPY	<Base Policy: PROPAGATE>		Propagate tags when they are copied.
4	STORE, LOAD, OP, OPIMM	<Base Policy: CLEAR>		Clear tags on output of operations and non-64-bit load/store.

Table 8.3: List of policies for return address protection.

8.3 Memory Compartmentalization

In large applications, a vulnerability in one component of a program can affect other components because there is no protection from accessing and modifying memory values in unrelated areas by the same process. With P-TAXI, a set of policies can be created to allow basic memory compartmentalization. For example, in the simple `malloc` implementation shown in Listing 8.1, P-TAXI can prevent other parts of the program that use this `malloc` implementation from accessing its metadata with the

set of policies shown in Table 8.4.

```
// ...
void *pmalloc_internal(size_t size) {
  malloc_ll *node = sbrk(sizeof(malloc_ll) + size);
  node->size = size;
  node->back = NULL;
  node->next = malloc_root;
  if (malloc_root != NULL) {
    malloc_root->back = node;
  }
  ptaxi_base_policy_settag_multi(MALLOC_TAGBIT, node, sizeof(
   malloc_ll), 1);
  malloc_root = node;
  void *res = ((void *) node) + sizeof(malloc_ll);
  return res;
}

void *pmalloc(size_t size) {
  ptaxi_policy_privilege_enter(MALLOC_TAGBIT);
  void *res = pmalloc_internal(size);
  ptaxi_policy_privilege_leave(MALLOC_TAGBIT);
  return res;
}
// ...
```

Listing 8.1: Malloc implementation that utilizes P-TAXI policies to enable memory compartmentalization.

8.4 Taint Tracking

Taint tracking is a mechanism for detecting the movement of data in a program by tracing flows of data. It can be used to detect security vulnerabilities, especially sensitive data leaks [63], [64].

P-TAXI can be used for basic taint tracking. Specifically, a set of policies can be created to track the propagation of an input or other tracked data. For example, in Listing 8.2, we can determine whether variable `D` is a result of computation that uses variable `A` with the set of polices shown in Table 8.5.

#	Filter		Action	Rationale
	Instruction Type	**Additional conditions**		
1	LOAD, LOAD64, STORE, STORE64	TAG(ARG1) = 1, PRIV = 0	BLOCK	Block non-privilege code from loading or overwriting sensitive area.
2	LOAD64, STORE64, COPY	<Base Policy: PROPAGATE>		Propagate tags when they are copied.
3	TAGCMD	<Base Policy: SETTAG>		Create SETTAG command.
4	—"—	<Base Policy: CLEARTAG>		Create CLEARTAG command.
5	—"—	<Base Policy: SETPRIV>		Create SETPRIV command.
6	—"—	<Base Policy: CLEARPRIV>		Create CLEARPRIV command.

Table 8.4: List of policies for memory compartmentalization.

```
// ...
uint64_t get_unfiltered_input() {
  uint64_t input = 42;
  ptaxi_base_policy_settag(TAINT_TAGBIT, (void *) (&input), 1);
  return input;
}

int main(int argc, char** argv) {
  uint64_t A = get_unfiltered_input();
  uint64_t B = 4;
  uint64_t C = B * 20;
  uint64_t D = A + 5;

  int s1 = ptaxi_base_policy_gettag(TAINT_TAGBIT, (void *) (&C));
  int s2 = ptaxi_base_policy_gettag(TAINT_TAGBIT, (void *) (&D));
  printf("TAG(C) = %d (should be 0), TAG(D) = %d (should be 1)\n",
   s1, s2);

  return 0;
}
```

Listing 8.2: Example code utilizing taint tracking.

#	Filter		Action	Rationale
	Instruction Type	**Additional conditions**		
1	LOAD, LOAD64, STORE STORE64, COPY, OPIMM	<Base Policy: PROPAGATE>		Propagate tags.
2	OP	<Base Policy: PROPAGATE> (but with either ARG1 or ARG2)		Since OP instructions have two operands, the result is tagged if either of inputs is tagged.
3	TAGCMD	<Base Policy: SETTAG>	-	Create SETTAG command.
4	—"—	<Base Policy: CLEARTAG>	-	Create CLEARTAG command.
5	—"—	<Base Policy: GETTAG>	-	Create GETTAG command.

Table 8.5: List of policies for taint tracking.

8.5 Stack Garbage Collection

P-TAXI provides a built-in stack garbage collection. Developers can utilize this by adding a policy with the `ACTION` field set to `GC` to P-TAXI. The common way to use this feature is to add a policy to do stack garbage collection on every function return.

8.6 Instruction Counting and Debugging

P-TAXI can be used to debug and measure the performance of a program in multiple ways. For example, a developer can add a policy to trap upon some specific instruction or to trap when some memory address is read or modified. P-TAXI also provides a built-in counter for each policy that can be used to count occurrences of instructions in specific conditions.

Chapter 9

Evaluation of P-TAXI

In this chapter, we evaluate the effectiveness of P-TAXI against multiple attacks and measure the performance with SPEC2006. We also provide a list of future works that can be conducted to improve P-TAXI.

9.1 Effectiveness Against Attacks

9.1.1 Code-Injection Attack

As we can see from Chapter 2, Data Execution Prevention is shown to be able to prevent code-injection attacks. A set of policies can be created to provide the same type of protection. However, a loader would need to be modified to mark instructions loaded from a binary with an appropriate tag.

Since the RISC-V architecture supports permissions in the page-table level, it is already possible to use DEP without P-TAXI. Therefore, it might be better to use the permission field in the page table in this case.

9.1.2 Code-Reuse Attack

To absolutely prevent code-reuse attacks, including Return-Oriented Programming, it is likely to be necessary to provide complete memory protection. However, many

defense schemes can be used to make it more difficult and impractical to attack a system. As shown in Chapter 8, tagging of return pointers can prevent an attacker from overwriting return pointers directly. If we can tag every data type correctly, it is also possible to use P-TAXI to provide additional protection. However, with the limitation of small tag per words, another protection scheme should also be used in combination with P-TAXI to increase the effectiveness of preventing code-reuse attacks.

9.1.3 Data-Oriented Programming

Data-Oriented Programming is a technique to attack a program without using any existing code on the memory. However, it uses primitive variables to allow an attacker to read or write from any location in memory, depending on vulnerabilities [65].

P-TAXI can be used to reduce the severity of the attack in several ways. First, compartmentalization would limit the scope of attacks because as the attacker would not be able to read or write to sensitive memory areas, except from a function that requires reading or writing to the area. Second, if tagging is done correctly for all pointers, an attacker might not be able to write to a pointer, which is one of the requirements of the attack. Third, if tagging is used for input filtering, a set of policies can be created to distinguish between validated data and raw input. A policy can be used to limit instructions that can be used with raw data.

9.1.4 Format String Attack

Format string attacks are caused by developers using an unfiltered input string directly as a format string. For example, if an unfiltered string is passed to the first argument of `printf`, an attacker would be able to utilize this string to read and write to arbitrary memory locations [66].

Similar to data-oriented programming, compartmentalization and data type tagging can help reducing the severity of format string attacks.

9.2 Performance

Since P-TAXI is programmable, it is difficult to measure objectively the performance of P-TAXI as it can vary depending on use cases and how the enforced set of policies is implemented. As such, we choose to measure the performance of P-TAXI for the set of policies that is used to enforce return address protection shown in Section 8.2. We decided to use SPEC2006 to provide a wide range of applications for the benchmark.

Because the implementation of P-TAXI is done solely on the Spike simulator, it is not possible to measure performance in actual CPU time. Therefore, we measure numbers of tag reads and writes to/from registers and memory instead. We did not consider using the tag cache, as such benchmark is already presented for TAXI. The result of the benchmark is shown in Table 9.1.

9.3 Future Works

The following are some of the ideas that can be implemented to improve P-TAXI:

- Improving data type tagging and integrating P-TAXI with other protection schemes to guarantee complete memory safety.
- Modifying P-TAXI and Linux kernel to enable on-the-fly insertion of policies. For example, a kernel can be modified so that traps resulting from a policy violation are automatically caught and sent back to the user-level side. This would allow user-level programs to add only necessary policies in each instance.
- Allowing user-level applications to specify the size of tag per words. This would allow user-level applications to have more flexibility in policy developments.
- Implementing the post-trap-handler approach, described in Section 7.4.3, for complex actions, including `GC`. The post-trap-handler approach would allow these complex actions to perform at the user level instead of the hardware level.

Test Name	Total # of instructions	Matched	# Read	# Write	With Read from ARG1			Without Read from ARG1		
					-	R OUT	RW OUT	-	R OUT	RW OUT
400.perlbench	2,687,020,357	72.50%	107.97%	2.32%	26.12%	36.80%	1.49%	1.50%	33.26%	0.83%
401.bzip2	34,637,103,431	71.50%	78.84%	1.12%	28.40%	63.15%	1.12%	0.10%	7.24%	0.00%
403.gcc	5,419,336,001	73.43%	111.27%	2.04%	24.99%	35.70%	1.47%	1.58%	35.69%	0.57%
410.bwaves	117,419,510,623	59.38%	67.86%	0.39%	39.85%	51.30%	0.37%	0.77%	7.69%	0.02%
416.gamess	242,675,586	53.02%	60.77%	0.91%	46.22%	45.22%	0.80%	0.77%	6.87%	0.11%
429.mcf	3,129,160,587	74.80%	114.98%	1.70%	24.96%	33.18%	1.67%	0.24%	39.92%	0.03%
433.milc	28,731,951,101	24.69%	31.42%	0.07%	74.62%	18.58%	0.07%	0.70%	6.04%	0.00%
435.gromacs	35,469,684,347	21.69%	25.85%	0.67%	77.49%	17.70%	0.66%	0.82%	3.33%	0.01%
436.cactusADM	13,497,373,910	22.00%	29.12%	0.14%	77.81%	14.93%	0.14%	0.18%	6.93%	0.00%
444.namd	48,640,271,482	29.02%	32.49%	0.09%	70.95%	25.49%	0.09%	0.03%	3.44%	0.00%
445.gobmk	69,972,013,926	73.68%	97.46%	1.08%	25.48%	49.97%	0.77%	0.84%	22.63%	0.31%
447.dealII	108,862,629,102	59.98%	73.32%	1.23%	38.45%	47.02%	1.19%	1.58%	11.73%	0.04%
450.soplex	95,242,541	59.62%	71.41%	0.80%	39.72%	47.77%	0.73%	0.66%	11.05%	0.08%
453.povray	5,968,418,895	57.11%	78.16%	1.05%	41.35%	36.83%	0.77%	1.53%	19.23%	0.28%
454.calculix	164,729,135	61.21%	74.06%	1.64%	38.03%	47.60%	1.51%	0.76%	11.96%	0.13%
456.hmmer	32,572,489,558	61.26%	68.70%	0.34%	38.11%	54.12%	0.34%	0.63%	6.80%	0.00%
458.sjeng	18,803,228,920	68.82%	81.71%	1.02%	30.57%	55.39%	0.99%	0.77%	12.25%	0.03%
459.GemsFDTD	7,466,145,344	50.11%	59.13%	0.67%	49.38%	41.16%	0.43%	0.51%	8.28%	0.23%
462.libquantum	315,468,561	66.57%	90.08%	0.09%	33.15%	43.26%	0.08%	0.28%	23.22%	0.01%
464.h264ref	106,500,405,338	69.90%	83.37%	0.54%	29.42%	56.25%	0.54%	1.00%	12.79%	0.00%
465.tonto	17,696,715,977	54.02%	62.82%	0.95%	44.90%	45.61%	0.67%	1.10%	7.44%	0.28%
470.lbm	26,346,119,630	50.36%	56.16%	0.34%	48.85%	45.01%	0.34%	0.79%	5.01%	0.00%
471.omnetpp	1,979,754,738	72.47%	116.22%	2.73%	24.25%	30.35%	1.64%	3.28%	39.38%	1.08%
473.astar	23,584,541,951	71.02%	94.30%	0.24%	28.38%	48.11%	0.23%	0.59%	22.68%	0.00%
481.wrf	71,296,120,342	32.55%	37.91%	0.80%	66.45%	27.41%	0.68%	1.10%	4.24%	0.12%
482.sphinx3	8,767,402,592	50.51%	63.94%	0.96%	48.90%	36.79%	0.86%	0.59%	12.75%	0.09%
483.xalancbmk	326,001,319	70.84%	113.92%	2.09%	26.97%	28.77%	1.18%	2.19%	39.99%	0.90%
Average	29,281,167,233	56.74%	73.45%	0.96%	42.36%	40.13%	0.77%	0.92%	15.62%	0.19%

Table 9.1: List of SPEC2006 tests used to estimate numbers of tag reads and writes with the enforcement of the P-TAXI return address protection policy set. All numbers except the total number of instructions are shown in percent of the total number of instructions. Test `434.zeusmp` is excluded as it resulted in segmentation fault. Test `437.leslie3d` is excluded as it did not complete within 10 hours in the modified Spike simulator.

Appendix A

P-TAXI Source Code

Due to space constraints, this appendix only displays the crucial parts of the P-TAXI modifications to the Spike simulator. Please see `https://github.com/riscv-mit/riscv-isa-sim/tree/ptaxi` *for the entire repository of the P-TAXI code base.*

A.1 Policy Definition (`ptaxi_common.h`)

```
#ifndef _PTAXI_COMMON_H
#define _PTAXI_COMMON_H

#include <stdint.h>

enum ptaxi_insn_type_t {
  PTAXI_INSN_TYPE_UNKNOWN,
  PTAXI_INSN_TYPE_LOAD,
  PTAXI_INSN_TYPE_LOAD64,
  PTAXI_INSN_TYPE_STORE,
  PTAXI_INSN_TYPE_STORE64,
  PTAXI_INSN_TYPE_COPY,
  PTAXI_INSN_TYPE_OP,
  PTAXI_INSN_TYPE_OPIMM,
  PTAXI_INSN_TYPE_JAL,
  PTAXI_INSN_TYPE_JALR,
  PTAXI_INSN_TYPE_RETURN,
  PTAXI_INSN_TYPE_TAGCMD,
  PTAXI_INSN_TYPE_TAGPOLICY,
};

enum {
  PTAXI_ACTION_CONTINUE = 0, // no action
```

```
  PTAXI_ACTION_ALLOW = 1,
  PTAXI_ACTION_BLOCK = 2,
  PTAXI_ACTION_GC = 4,
  PTAXI_ACTION_CALL = 8,
  PTAXI_ACTION_DEBUG_LINE = 16,
  PTAXI_ACTION_DEBUG_DETAIL = 32,
  PTAXI_ACTION_GETTAG = 64,
};

typedef uint8_t ptaxi_action_t;

struct ptaxi_policy_t {
  // Filter
  enum ptaxi_insn_type_t insn_type :8;
  uint8_t rs1_mask;
  uint8_t rs1_match;
  uint8_t rs2_mask;
  uint8_t rs2_match;
  uint8_t rs1val_mask;
  uint8_t rs1val_match;
  uint8_t rs2val_mask;
  uint8_t rs2val_match;
  uint8_t tag_arg1_mask;
  uint8_t tag_arg1_match;
  uint8_t tag_arg2_mask;
  uint8_t tag_arg2_match;
  uint8_t tag_out_mask;
  uint8_t tag_out_match;
  uint8_t priv_mask;
  uint8_t priv_match;

  // Action
  ptaxi_action_t action :8;
  uint8_t tag_out_set;
  uint8_t tag_out_tomodify;
  uint8_t priv_set;
  uint8_t priv_tomodify;
  uint8_t ignore_count;
};


union ptaxi_policy_serialized {
  struct ptaxi_policy_t policy;
  struct ptaxi_policy_serialized_result {
    uint64_t a;
    uint64_t b;
    uint64_t c;
  } regs;
};

#endif
```

Listing A.1: `ptaxi_common.h`

A.2 P-TAXI Simulator

A.2.1 Header File (ptaxisim.h)

```
// See LICENSE for license details.

#ifndef _RISCV_PTAXI_SIM_H
#define _RISCV_PTAXI_SIM_H

#include "decode.h"
#include "mmu.h"
#include "processor.h"
#include <vector>
#include <utility>
#include "ptaxi_common.h"

struct ptaxi_policy_context_t {
  ptaxi_policy_t policy;
  size_t match_count;
};
struct ptaxi_context_state_t {
  std::vector<ptaxi_policy_context_t> policy_contexts;
  bool is_enable;
  uint8_t priv_bits;
  uint64_t lowest_sp_addr;
};

struct ptaxi_benchmark_counters {
  uint64_t insns;
  uint64_t match_insns;
  uint64_t tag_read;
  uint64_t tag_write;
  uint64_t needs[16]; // 16 bits, RARG1/RARG2/ROUT/WOUT
};

#define TAG_RET_FROM_JAL 1
#define TAG_RET_FROM_MEM 2

enum insn_var_type_t {
  INSN_OUT, INSN_ARG1, INSN_ARG2,
};

class ptaxi_sim_t {
public:
  ptaxi_sim_t();
  reg_t execute_insn(processor_t *p, reg_t pc, insn_fetch_t fetch);
  void add_policy(processor_t *p, uint64_t a, uint64_t b, uint64_t c
   );
  void run_tag_command(processor_t *p, uint64_t cmd);
  void start_benchmark(processor_t *p);
```

```
46   void stop_benchmark(processor_t *p);
47 private:
48   void print_policies(size_t context_id);
49   ptaxi_insn_type_t get_insn_type(insn_t insn);
50   size_t get_ptaxi_context_id(processor_t *p, bool add_if_needed);
51   std::pair<ptaxi_action_t, int> determine_ptaxi_action(processor_t
      *p, insn_t insn, reg_t pc);
52   uint8_t get_or_set_tag(processor_t *p, insn_t insn, reg_t pc,
      ptaxi_insn_type_t insn_type,
53       insn_var_type_t var_type, bool set_tag, uint8_t tag_val);
54   uint8_t load_tag_from_mem(processor_t *p, uint64_t addr, uint8_t
      rm);
55   void store_tag_to_mem(processor_t *p, uint64_t addr, uint8_t rm,
      uint64_t val);
56   tagged_reg_t v;
57   // states[0] is a default policy template.
58   std::vector<ptaxi_context_state_t> states;
59   bool benchmark_mode = false;
60   ptaxi_benchmark_counters counters;
61 };
62 #endif
```

Listing A.2: ptaxisim.h

A.2.2 Source File (ptaxisim.cc)

```
1  #include "ptaxisim.h"
2
3  #include "mmu.h"
4  #include "disasm.h"
5  #include "decode.h"
6  #include <vector>
7
8  // From https://stackoverflow.com/questions/3219393/stdlib-and-
      colored-output-in-c
9  #define ANSI_COLOR_RED     "\x1b[31m"
10 #define ANSI_COLOR_GREEN   "\x1b[32m"
11 #define ANSI_COLOR_YELLOW  "\x1b[33m"
12 #define ANSI_COLOR_BLUE    "\x1b[34m"
13 #define ANSI_COLOR_MAGENTA "\x1b[35m"
14 #define ANSI_COLOR_CYAN    "\x1b[36m"
15 #define ANSI_COLOR_RESET   "\x1b[0m"
16
17 #define OPCODE_LOAD      (0b0000011)
18 #define OPCODE_LOADFP    (0b0000111)
19 #define OPCODE_MISCMEM   (0b0001111)
20 #define OPCODE_OPIMM     (0b0010011)
21 #define OPCODE_AUIPC     (0b0010111)
22 #define OPCODE_OPIMM32   (0b0011011)
23 #define OPCODE_STORE     (0b0100011)
24 #define OPCODE_STOREFP   (0b0100111)
25 #define OPCODE_AMO       (0b0101111)
```

```
26 #define OPCODE_OP         (0b0110011)
27 #define OPCODE_LUI        (0b0110111)
28 #define OPCODE_OP32       (0b0111011)
29 #define OPCODE_MADD       (0b1000011)
30 #define OPCODE_MSUB       (0b1000111)
31 #define OPCODE_NMSUB      (0b1001011)
32 #define OPCODE_NMADD      (0b1001111)
33 #define OPCODE_OPFP       (0b1010011)
34 #define OPCODE_BRANCH     (0b1100011)
35 #define OPCODE_JALR       (0b1100111)
36 #define OPCODE_JAL        (0b1101111)
37 #define OPCODE_SYSTEM     (0b1110011)
38
39 #define OPCODE_TAGCMD     (0b0001011)
40 #define OPCODE_TAGPOLICY  (0b0101011)
41
42 #define REG_SP 2
43
44 #define TAG_RET_FROM_JAL 1
45 #define TAG_RET_FROM_MEM 2
46 #define SR_TAG_SHIFT 9
47 #define PTAXI_DEBUG_MODE_CONTEXT_ID 42 // Any number > 0 is fine
       here, just for debugging purpose.
48
49 // Adapted from https://stackoverflow.com/questions/1941307/c-debug-
       print-macros
50 #define PTAXI_VERBOSE
51 //#define PTAXI_DEBUG
52 #ifdef PTAXI_VERBOSE
53 #define DPRINTF(fmt, args...) printf(fmt, ## args)
54 #else
55 #define DPRINTF(fmt, args...)
56 #endif
57
58 #ifdef PTAXI_DEBUG
59 #define DDPRINTF(fmt, args...) printf(fmt, ## args)
60 #else
61 #define DDPRINTF(fmt, args...)
62 #endif
63
64 ptaxi_sim_t::ptaxi_sim_t() {
65   struct ptaxi_context_state_t default_state;
66   default_state.is_enable = false;
67   default_state.priv_bits = 0;
68   default_state.lowest_sp_addr = 0;
69   states.push_back(default_state);
70 }
71
72 ptaxi_insn_type_t ptaxi_sim_t::get_insn_type(insn_t insn) {
73   switch (insn.opcode()) {
74   case OPCODE_LOAD:
75     if (insn.rm() == 3) {
76       return PTAXI_INSN_TYPE_LOAD64;
```

```
77      } else {
78        return PTAXI_INSN_TYPE_LOAD;
79      }
80    case OPCODE_STORE:
81      if (insn.rm() == 3) {
82        return PTAXI_INSN_TYPE_STORE64;
83      } else {
84        return PTAXI_INSN_TYPE_STORE;
85      }
86    case OPCODE_OP:
87      return PTAXI_INSN_TYPE_OP;
88    case OPCODE_OPIMM:
89      if (insn.rm() == 0 && insn.i_imm() == 0) {
90        return PTAXI_INSN_TYPE_COPY;
91      }
92      return PTAXI_INSN_TYPE_OPIMM;
93    case OPCODE_JAL:
94      return PTAXI_INSN_TYPE_JAL;
95    case OPCODE_JALR:
96      // rs1 == X_RA (X_RA = 1)
97      if (insn.i_imm() == 0 && insn.rs1() == 1 && insn.rm() == 0 &&
       insn.rd() == 0) {
98        return PTAXI_INSN_TYPE_RETURN;
99      }
100     return PTAXI_INSN_TYPE_JALR;
101   case OPCODE_TAGCMD:
102     return PTAXI_INSN_TYPE_TAGCMD;
103   case OPCODE_TAGPOLICY:
104     return PTAXI_INSN_TYPE_TAGPOLICY;
105
106   default:
107     return PTAXI_INSN_TYPE_UNKNOWN;
108   }
109 }
110
111 size_t ptaxi_sim_t::get_ptaxi_context_id(processor_t *p, bool
       add_if_needed) {
112   size_t context_id;
113   if (benchmark_mode) {
114     context_id = PTAXI_DEBUG_MODE_CONTEXT_ID;
115   } else {
116     context_id = (p->get_pcr(CSR_STATUS) & SR_TAG) >> SR_TAG_SHIFT;
117   }
118   if (add_if_needed && context_id == 0) {
119     reg_t old = p->get_pcr(CSR_STATUS);
120     context_id = states.size();
121     if (context_id >= (1 << 7)) {
122       DPRINTF("Context ID Full...\n");
123       return 0;
124     }
125     p->set_pcr(CSR_STATUS, old | (context_id << SR_TAG_SHIFT));
126   }
127   while (context_id >= states.size()) {
```

```
128     states.push_back(states[0]);
129   }
130   return context_id;
131 }
132
133 void print_insn(processor_t *p, const char *str, insn_t insn) {
134   disassembler_t* disas = p->get_disassembler();
135   printf("\x1b[32m%s: %-25s", str, disas->disassemble(insn).c_str())
      ;
136   printf("RS1: %2lu, RS2: %2lu, IMM: %8ld, RS1VAL: %8lu (0x%8lx),
      RS2VAL: %8lu (0x%8lx)\x1b[0m\n",
137       insn.rs1(), insn.rs2(), insn.i_imm(), RS1, RS1, RS2, RS2);
138 }
139
140 std::pair<ptaxi_action_t, int> ptaxi_sim_t::determine_ptaxi_action(
      processor_t *p, insn_t insn,
141     reg_t pc) {
142   size_t context_id = get_ptaxi_context_id(p, false);
143   if (context_id == 0 || !states[context_id].is_enable ||
      IS_SUPERVISOR) {
144     return std::make_pair(0, -2);
145   }
146
147   ptaxi_insn_type_t insn_type = get_insn_type(insn);
148   ptaxi_action_t action = 0;
149   uint8_t tag_arg1 = 0, tag_arg2 = 0, tag_out = 0, tag_out_updated =
      0;
150   bool is_load_tag_arg1 = false, is_load_tag_arg2 = false,
      is_load_tag_out = false;
151   bool has_match = false;
152   size_t i;
153
154   for (i = 0; i < states[context_id].policy_contexts.size(); i++) {
155     struct ptaxi_policy_t policy = states[context_id].
      policy_contexts[i].policy;
156     bool match = (insn_type == policy.insn_type);
157     if (match && policy.rs1_mask) {
158       match = match && ((insn.rs1() & policy.rs1_mask) == policy.
      rs1_match);
159     }
160     if (match && policy.rs2_mask) {
161       match = match && ((insn.rs2() & policy.rs2_mask) == policy.
      rs2_match);
162     }
163
164     if (match && policy.priv_mask) {
165       match = match && ((states[context_id].priv_bits & policy.
      priv_mask) == policy.priv_match);
166     }
167
168     if (match && policy.rs1val_mask) {
169       match = match && ((RS1 & policy.rs1val_mask) == policy.
      rs1val_match);
```

```
170     }
171     if (match && policy.rs2val_mask) {
172       match = match && ((RS2 & policy.rs2val_mask) == policy.
    rs2val_match);
173     }
174
175     if (match && policy.tag_arg1_mask) {
176       if (!is_load_tag_arg1) {
177         is_load_tag_arg1 = true;
178         tag_arg1 = get_or_set_tag(p, insn, pc, insn_type, INSN_ARG1,
     false, 0);
179         if (benchmark_mode) {
180           counters.tag_read++;
181         }
182       }
183       match = match && ((tag_arg1 & policy.tag_arg1_mask) == policy.
    tag_arg1_match);
184     }
185     if (match && policy.tag_arg2_mask) {
186       if (!is_load_tag_arg2) {
187         is_load_tag_arg2 = true;
188         tag_arg2 = get_or_set_tag(p, insn, pc, insn_type, INSN_ARG2,
     false, 0);
189         if (benchmark_mode) {
190           counters.tag_read++;
191         }
192       }
193       match = match && ((tag_arg2 & policy.tag_arg2_mask) == policy.
    tag_arg2_match);
194     }
195
196     if (match && (policy.tag_out_mask || policy.tag_out_tomodify)) {
197       if (!is_load_tag_out) {
198         is_load_tag_out = true;
199         tag_out = get_or_set_tag(p, insn, pc, insn_type, INSN_OUT,
    false, 0);
200         tag_out_updated = tag_out;
201         if (benchmark_mode) {
202           counters.tag_read++;
203         }
204       }
205       match = match && ((tag_out & policy.tag_out_mask) == policy.
    tag_out_match);
206     }
207
208     if (match) {
209       has_match = true;
210       struct ptaxi_policy_context_t &policy_context = states[
    context_id].policy_contexts[i];
211       policy_context.match_count++;
212       if (policy_context.match_count <= policy_context.policy.
    ignore_count) {
213         continue;
```

```
214       }
215       tag_out_updated = ((tag_out_updated & (~policy.
    tag_out_tomodify)) | policy.tag_out_set);
216       if (policy.priv_tomodify) {
217         states[context_id].priv_bits = ((states[context_id].
    priv_bits & (~policy.priv_tomodify))
218             | policy.priv_set);
219         DPRINTF("Priv Bits set to %d\n", (int) states[context_id].
    priv_bits);
220       }
221
222
223       action |= policy.action;
224       if (policy.action == PTAXI_ACTION_BLOCK || policy.action ==
    PTAXI_ACTION_ALLOW) {
225         break;
226       }
227     }
228   }
229
230   bool real_tag_update = false;
231   if (is_load_tag_out && (tag_out != tag_out_updated)) {
232     real_tag_update = true;
233     get_or_set_tag(p, insn, pc, insn_type, INSN_OUT, true,
    tag_out_updated);
234   }
235
236   if (benchmark_mode) {
237     uint8_t bits = (((uint8_t) is_load_tag_arg1) << 3) + (((uint8_t)
     is_load_tag_arg2) << 2) +
238         (((uint8_t) is_load_tag_out) << 1) + (uint8_t)
    real_tag_update;
239     if (real_tag_update) {
240       counters.tag_write++;
241     }
242     counters.insns++;
243     counters.needs[bits]++;
244     if (has_match) {
245       counters.match_insns++;
246     }
247   }
248
249   return std::make_pair(action, i);
250 }
251
252 reg_t ptaxi_sim_t::execute_insn(processor_t *p, reg_t pc,
    insn_fetch_t fetch) {
253   insn_t insn = fetch.insn;
254   uint64_t before_tag_val = 0;
255   ptaxi_insn_type_t insn_type = get_insn_type(insn);
256   if (insn_type == PTAXI_INSN_TYPE_TAGCMD && insn.rd() != 0) {
257     before_tag_val = TAG_S2;
258   }
```

```
259
260   std::pair<ptaxi_action_t, int> paction = determine_ptaxi_action(p,
        insn, pc);
261   ptaxi_action_t action = paction.first;
262   disassembler_t* disas = p->get_disassembler();
263
264   if (!benchmark_mode) {
265     if (action & PTAXI_ACTION_DEBUG_LINE) {
266       printf(ANSI_COLOR_CYAN "%p: %-25s DEBUG\n" ANSI_COLOR_RESET, (
      void *) pc,
267           disas->disassemble(insn).c_str());
268     }
269
270     if (action & PTAXI_ACTION_DEBUG_DETAIL) {
271       size_t context_id = get_ptaxi_context_id(p, true);
272       printf(ANSI_COLOR_MAGENTA "PTAXI_ACTION_DEBUG_DETAIL: %s\n",
273           disas->disassemble(insn).c_str());
274       printf("PC: %lx, Exit Rule: %d, Context ID: %lu\n", pc,
      paction.second, context_id);
275       print_insn(p, "INSN", fetch.insn);
276       print_policies(context_id);
277       printf(ANSI_COLOR_RESET);
278     }
279
280     if (action & PTAXI_ACTION_BLOCK) {
281       size_t context_id = get_ptaxi_context_id(p, true);
282
283       printf(ANSI_COLOR_MAGENTA "PTAXI_ACTION_BLOCK: %s\n"
      ANSI_COLOR_RESET,
284           disas->disassemble(insn).c_str());
285       print_policies(context_id);
286       throw trap_tag_violation();
287       return pc;
288     }
289   }
290
291   if (action & PTAXI_ACTION_GC) {
292     size_t context_id = get_ptaxi_context_id(p, false);
293     uint64_t cur_sp = STATE.XPR[REG_SP];
294     uint64_t lowest = states[context_id].lowest_sp_addr;
295     DDPRINTF(ANSI_COLOR_MAGENTA "%10p: %-25s GCSTAR (--) = %p %p\n"
      ANSI_COLOR_RESET, (void *) pc,
296       disas->disassemble(insn).c_str(), (void *) cur_sp, (void *)
      lowest);
297     uint64_t clean_from = lowest - 8;
298     uint64_t clean_to = cur_sp - 8;
299     uint64_t clean_at;
300     for(clean_at = clean_from; clean_at < clean_to; clean_at += 8) {
301       MMU.store_tagged_uint64(clean_at, 0, 0);
302     }
303
304     DDPRINTF(ANSI_COLOR_GREEN "CLEAN FROM %p to %p\n"
      ANSI_COLOR_RESET, (void *) clean_from,
```

```
305         (void *) clean_to);
306
307     states[context_id].lowest_sp_addr = cur_sp;
308   }
309
310
311   if (insn_type == PTAXI_INSN_TYPE_TAGCMD && insn.rd() != 0) {
312     if (action & PTAXI_ACTION_GETTAG) {
313       DDPRINTF(ANSI_COLOR_CYAN "%10p: %-25s GETTAG (%2d) = %d\n"
    ANSI_COLOR_RESET, (void *) pc,
314           disas->disassemble(insn).c_str(), (int) insn.rs2(), (int)
    before_tag_val);
315       WRITE_RD(before_tag_val);
316     } else {
317       WRITE_RD(RS2);
318     }
319   }
320
321   reg_t res = fetch.func(p, insn, pc);
322
323   if(insn.rd() == REG_SP && !IS_SUPERVISOR) {
324     size_t context_id = get_ptaxi_context_id(p, false);
325     if (context_id != 0) {
326       uint64_t cur_sp = STATE.XPR[REG_SP];
327       DDPRINTF(ANSI_COLOR_CYAN "%10p: %-25s MODISP (--) = %p\n"
    ANSI_COLOR_RESET, (void *) pc,
328           disas->disassemble(insn).c_str(), (void *) cur_sp);
329       if (cur_sp < states[context_id].lowest_sp_addr || states[
    context_id].lowest_sp_addr == 0) {
330         states[context_id].lowest_sp_addr = cur_sp;
331         DDPRINTF(ANSI_COLOR_BLUE "%10p: %-25s LOWEST (--) = %p\n"
    ANSI_COLOR_RESET, (void *) pc,
332                 disas->disassemble(insn).c_str(), (void *) cur_sp)
    ;
333       }
334     }
335   }
336
337   return res;
338 }
339
340 void ptaxi_sim_t::print_policies(size_t context_id) {
341   printf("Policy Count: %lu\n------\n", states[context_id].
    policy_contexts.size());
342   for (size_t i = 0; i < states[context_id].policy_contexts.size();
    i++) {
343     struct ptaxi_policy_context_t policy_context = states[context_id
    ].policy_contexts[i];
344     printf("%3lu |%5d |%3d%3d |%3lu\n", i, (int) policy_context.
    policy.insn_type,
345         (int) policy_context.policy.rs1val_match, (int)
    policy_context.policy.action,
346         policy_context.match_count);
```

```
347   }
348   printf("------\n");
349 }
350
351 void ptaxi_sim_t::add_policy(processor_t *p, uint64_t a, uint64_t b,
        uint64_t c) {
352   size_t context_id = get_ptaxi_context_id(p, true);
353   union ptaxi_policy_serialized ps;
354   ps.regs.a = a;
355   ps.regs.b = b;
356   ps.regs.c = c;
357
358   struct ptaxi_policy_context_t policy_context;
359   policy_context.policy = ps.policy;
360   policy_context.match_count = 0;
361   states[context_id].policy_contexts.push_back(policy_context);
362 }
363
364 void ptaxi_sim_t::run_tag_command(processor_t *p, uint64_t cmd) {
365   size_t context_id = get_ptaxi_context_id(p, true);
366   if (cmd == 0) {
367     DPRINTF(ANSI_COLOR_CYAN "Enforcing.. Context Id = %d\n"
       ANSI_COLOR_RESET, (int) context_id);
368     states[context_id].is_enable = true;
369   } else {
370     DPRINTF(ANSI_COLOR_YELLOW "TAG COMMAND %lu\n" ANSI_COLOR_RESET,
       cmd);
371   }
372   #ifdef PTAXI_VERBOSE
373     print_policies(context_id);
374   #endif
375 }
376
377 uint8_t ptaxi_sim_t::get_or_set_tag(processor_t *p, insn_t insn,
       reg_t pc,
378     ptaxi_insn_type_t insn_type, insn_var_type_t var_type, bool
       set_tag, uint8_t tag_val) {
379   bool is_invalid = false, is_mem = false;
380   uint8_t reg;
381   uint64_t addr;
382
383   switch (insn_type) {
384   case PTAXI_INSN_TYPE_LOAD64: // arg1 = MEM, arg2 = N/A, out = REG
385   case PTAXI_INSN_TYPE_LOAD:
386     if (var_type == INSN_ARG1) {
387       is_mem = true;
388       addr = RS1+ insn.i_imm();
389     } else if (var_type == INSN_ARG2) {
390       is_invalid = true;
391     } else {
392       reg = insn.rd();
393     }
394     break;
```

```
395     case PTAXI_INSN_TYPE_STORE64: // arg1 = REG, arg2 = N/A, out =
    MEM
396     case PTAXI_INSN_TYPE_STORE:
397     if (var_type == INSN_ARG1) {
398       reg = insn.rs2();
399     } else if (var_type == INSN_ARG2) {
400       is_invalid = true;
401     } else {
402       is_mem = true;
403       addr = RS1 + insn.s_imm();
404     }
405     break;
406     case PTAXI_INSN_TYPE_TAGCMD:
407     case PTAXI_INSN_TYPE_OP: // arg1 = REG1, arg2 = REG2, out =
    REGOUT
408     if (var_type == INSN_ARG1) {
409       reg = insn.rs1();
410     } else if (var_type == INSN_ARG2) {
411       reg = insn.rs2();
412     } else {
413       reg = insn.rd();
414     }
415     break;
416     case PTAXI_INSN_TYPE_OPIMM: // arg1 = REG1, arg2 = N/A, out =
    REGOUT
417     case PTAXI_INSN_TYPE_COPY:
418     if (var_type == INSN_ARG1) {
419       reg = insn.rs1();
420     } else if (var_type == INSN_ARG2) {
421       is_invalid = true;
422     } else {
423       reg = insn.rd();
424     }
425     break;
426     case PTAXI_INSN_TYPE_JAL: // arg1 = TARGET, arg2 = n/a, arg3 =
    REGOUT
427     if (var_type == INSN_ARG1) {
428       /*is_mem = true;
429        addr = pc + insn.uj_imm();*/
430       is_invalid = true;
431     } else if (var_type == INSN_ARG2) {
432       is_invalid = true;
433     } else {
434       reg = insn.rd();
435     }
436     break;
437     case PTAXI_INSN_TYPE_JALR: // arg1 = REG1, arg2 = TARGET, arg3 =
     REGOUT
438     case PTAXI_INSN_TYPE_RETURN:
439     if (var_type == INSN_ARG1) {
440       reg = insn.rs1();
441     } else if (var_type == INSN_ARG2) {
442       is_mem = true;
```

```
443       addr = (RS1 + insn.i_imm()) & ~reg_t(1);
444     } else {
445       reg = insn.rd();
446     }
447     break;
448     default:
449     is_invalid = true;
450     break;
451   }
452
453   if (is_invalid) {
454     DPRINTF("get_or_set_tag: ISINVALID TRAP!\n");
455     DPRINTF("GET OR SET TAG %lx %d %d %d\n", insn.bits(), (int)
      insn_type, (int) var_type,
456         (int) set_tag);
457     throw trap_tag_violation();
458     return 0;
459   }
460   disassembler_t* disas = p->get_disassembler();
461
462   if (is_mem) {
463     if (set_tag) {
464       store_tag_to_mem(p, addr, insn.rm(), tag_val);
465       DDPRINTF(ANSI_COLOR_CYAN "%10p: %-25s SETMEM (%p) = %d\n"
      ANSI_COLOR_RESET, (void *) pc,
466           disas->disassemble(insn).c_str(), (void *) addr, (int)
      tag_val);
467
468     } else {
469       uint8_t tag_val_from_mem = load_tag_from_mem(p, addr, insn.rm
      ());
470       DDPRINTF(ANSI_COLOR_CYAN "%10p: %-25s LOADTG (%p) = %d\n"
      ANSI_COLOR_RESET, (void *) pc,
471           disas->disassemble(insn).c_str(), addr, (int)
      tag_val_from_mem);
472       return tag_val_from_mem;
473     }
474   } else {
475     if (reg == 0) {
476       return 0;
477     }
478     if (set_tag) {
479       DDPRINTF(ANSI_COLOR_CYAN "%10p: %-25s SETREG (%2d) = %d\n"
      ANSI_COLOR_RESET, (void *) pc,
480           disas->disassemble(insn).c_str(), (int) reg, (int) tag_val
      );
481       STATE.XPR.write_tag(reg, tag_val);
482     } else {
483       return STATE.XPR.read_tag(reg);
484     }
485   }
486   return 0;
487 }
```

```
488
489 uint8_t ptaxi_sim_t::load_tag_from_mem(processor_t *p, uint64_t addr
    , uint8_t rm) {
490   switch (rm) {
491   case 0: // LB
492     return MMU.load_tag_only_int8(addr);
493   case 1: // LH
494     return MMU.load_tag_only_int16(addr);
495   case 2: // LW
496     return MMU.load_tag_only_int32(addr);
497   case 3: // LD
498     return MMU.load_tag_only_int64(addr);
499   case 4: // LBU
500     return MMU.load_tag_only_uint8(addr);
501   case 5: // LHU
502     return MMU.load_tag_only_uint16(addr);
503   case 6: // LWU
504     return MMU.load_tag_only_uint32(addr);
505   default:
506     DPRINTF("get_or_set_tag: ISINVALID TRAP2!");
507     throw trap_tag_violation();
508     return 0;
509   }
510 }
511
512 void ptaxi_sim_t::store_tag_to_mem(processor_t *p, uint64_t addr,
    uint8_t rm, uint64_t val) {
513   switch (rm) {
514   case 0: // SB
515     MMU.store_tag_only_uint8(addr, val);
516     break;
517   case 1: // SH
518     MMU.store_tag_only_uint16(addr, val);
519     break;
520   case 2: // SW
521     MMU.store_tag_only_uint32(addr, val);
522     break;
523   case 3: // SD
524     MMU.store_tag_only_uint64(addr, val);
525     break;
526   default:
527     DPRINTF("get_or_set_tag: ISINVALID TRAP3!");
528     throw trap_tag_violation();
529     break;
530   }
531 }
532
533 void ptaxi_sim_t::start_benchmark(processor_t *p) {
534   if (benchmark_mode) {
535     return;
536   }
537   DPRINTF(ANSI_COLOR_GREEN "Start Benchmark..\n" ANSI_COLOR_RESET);
538   memset(&counters, 0, sizeof(counters));
```

```
539   benchmark_mode = true;
540 }
541
542 void ptaxi_sim_t::stop_benchmark(processor_t *p) {
543   if (!benchmark_mode) {
544     return;
545   }
546   DPRINTF(ANSI_COLOR_GREEN "Stop Benchmark..\n" ANSI_COLOR_RESET);
547   print_policies(PTAXI_DEBUG_MODE_CONTEXT_ID);
548   printf("RESULT,%lu,%lu,%lu,%lu", counters.insns, counters.
       match_insns, counters.tag_read, counters.tag_write);
549   for(int i = 0; i < 16; i++) {
550     printf(",%lu", counters.needs[i]);
551   }
552   printf("\n");
553   benchmark_mode = false;
554 }
```

Listing A.3: ptaxisim.cc

A.3 User-level Libraries

A.3.1 Basic Interface (ptaxi_user.h)

```
1  #ifndef _PTAXI_USER_H
2  #define _PTAXI_USER_H
3
4  #include <stdint.h>
5  #include <string.h>
6
7  #include "ptaxi_common.h"
8
9  // Use macro so that we don't have to set previous return pointers.
10 void ptaxi_tag_command(code) {
11   __asm__("tagcmd zero,%0,zero" ::"r"(code));
12
13 }
14
15 void ptaxi_enforce_policy() {
16   ptaxi_tag_command(0);
17 }
18
19 void ptaxi_add_raw_policy(uint64_t a, uint64_t b, uint64_t c) {
20   __asm__ volatile ("tagpolicy %2,%0,%1" ::"r"(a), "r"(b), "r"(c));
21 }
22
23 void ptaxi_add_policy(struct ptaxi_policy_t policy) {
24   union ptaxi_policy_serialized ps;
25   ps.policy = policy;
26   ptaxi_add_raw_policy(ps.regs.a, ps.regs.b, ps.regs.c);
```

```
}

#endif
```

Listing A.4: `ptaxi_user.h`

A.3.2 Header File for Inclusion by User-Level Applications (`ptaxi.h`)

```
#ifndef _PTAXI_H
#define _PTAXI_H

#include "ptaxi_common.h"
#include "ptaxi_user.h"
#include "ptaxi_policy_debug_call.h"
#include "ptaxi_policy_return_address.h"
#include "ptaxi_policy_privilege.h"
#include "ptaxi_policy_gc.h"

#endif
```

Listing A.5: `ptaxi.h`

Appendix B

Policy Source Code and Test Cases

Policy source code and test cases are also available at `https://github.com/riscv-mit/ptaxi-policies`.

B.1 Policies

B.1.1 Base Policies

```
#ifndef _PTAXI_BASE_POLICY_H
#define _PTAXI_BASE_POLICY_H

#define PTAXI_TAGCMD_PREFIX_GETTAG    100
#define PTAXI_TAGCMD_PREFIX_SETTAG    110
#define PTAXI_TAGCMD_PREFIX_CLEARTAG  120
#define PTAXI_TAGCMD_PREFIX_SETPRIV   130
#define PTAXI_TAGCMD_PREFIX_CLEARPRIV 140

uint8_t log2bit(uint8_t tagbit) {
  uint8_t out = 0;
  while(tagbit != 0) {
    out++;
    tagbit = (tagbit >> 1);
  }
  return out;
}
void ptaxi_base_policy_propatgate_by_type(uint8_t tagbit, enum
    ptaxi_insn_type_t insn_type,
    uint8_t arg1, uint8_t arg2) {
  struct ptaxi_policy_t default_policy, policy;
```

```
21   memset(&default_policy, 0, sizeof(default_policy));
22   default_policy.tag_out_tomodify = tagbit;
23   default_policy.insn_type = insn_type;
24
25   if (arg1) {
26     policy = default_policy;
27     policy.tag_arg1_mask = tagbit;
28     policy.tag_arg1_match = tagbit;
29     policy.tag_out_set = tagbit;
30     ptaxi_add_policy(policy);
31   }
32
33   if (arg2) {
34     policy = default_policy;
35     policy.tag_arg2_mask = tagbit;
36     policy.tag_arg2_match = tagbit;
37     policy.tag_out_set = tagbit;
38     ptaxi_add_policy(policy);
39   }
40
41   policy = default_policy;
42   policy.tag_out_set = 0;
43   if (arg1) {
44     policy.tag_arg1_mask = tagbit;
45     policy.tag_arg1_match = 0;
46   }
47   if (arg2) {
48     policy.tag_arg2_mask = tagbit;
49     policy.tag_arg2_match = 0;
50   }
51   ptaxi_add_policy(policy);
52 }
53
54
55 void ptaxi_base_policy_clear_by_type(uint8_t tagbit, enum
       ptaxi_insn_type_t insn_type) {
56   struct ptaxi_policy_t policy;
57   memset(&policy, 0, sizeof(policy));
58   policy.tag_out_tomodify = tagbit;
59   policy.insn_type = insn_type;
60   policy.tag_out_set = 0;
61   ptaxi_add_policy(policy);
62 }
63
64
65 void ptaxi_base_policy_create_settag(uint8_t tagbit) {
66   struct ptaxi_policy_t policy;
67   memset(&policy, 0, sizeof(policy));
68   policy.insn_type = PTAXI_INSN_TYPE_TAGCMD;
69   policy.rs1val_mask = 0b11111111;
70   policy.rs1val_match = PTAXI_TAGCMD_PREFIX_SETTAG + log2bit(tagbit)
       ;
71   policy.tag_out_tomodify = tagbit;
```

```
72   policy.tag_out_set = tagbit;
73   ptaxi_add_policy(policy);
74 }
75
76 void ptaxi_base_policy_create_cleartag(uint8_t tagbit) {
77   struct ptaxi_policy_t policy;
78   memset(&policy, 0, sizeof(policy));
79   policy.tag_out_tomodify = tagbit;
80   policy.insn_type = PTAXI_INSN_TYPE_TAGCMD;
81   policy.rs1val_mask = 0b11111111;
82   policy.rs1val_match = PTAXI_TAGCMD_PREFIX_CLEARTAG + log2bit(
       tagbit);
83   policy.tag_out_set = 0;
84   ptaxi_add_policy(policy);
85 }
86
87 void ptaxi_base_policy_create_gettag(uint8_t tagbit) {
88   struct ptaxi_policy_t policy;
89   memset(&policy, 0, sizeof(policy));
90   policy.tag_out_tomodify = tagbit;
91   policy.insn_type = PTAXI_INSN_TYPE_TAGCMD;
92   policy.rs1val_mask = 0b11111111;
93   policy.rs1val_match = PTAXI_TAGCMD_PREFIX_GETTAG + log2bit(tagbit)
       ;
94   policy.tag_out_set = 0;
95   policy.action = PTAXI_ACTION_GETTAG;
96   ptaxi_add_policy(policy);
97 }
98
99 void ptaxi_base_policy_create_setpriv(uint8_t tagbit) {
100  struct ptaxi_policy_t policy;
101  memset(&policy, 0, sizeof(policy));
102  policy.insn_type = PTAXI_INSN_TYPE_TAGCMD;
103  policy.rs1val_mask = 0b11111111;
104  policy.rs1val_match = PTAXI_TAGCMD_PREFIX_SETPRIV + log2bit(tagbit
       );
105  policy.priv_tomodify = tagbit;
106  policy.priv_set = tagbit;
107  ptaxi_add_policy(policy);
108 }
109
110 void ptaxi_base_policy_create_clearpriv(uint8_t tagbit) {
111  struct ptaxi_policy_t policy;
112  memset(&policy, 0, sizeof(policy));
113  policy.insn_type = PTAXI_INSN_TYPE_TAGCMD;
114  policy.rs1val_mask = 0b11111111;
115  policy.rs1val_match = PTAXI_TAGCMD_PREFIX_CLEARPRIV + log2bit(
       tagbit);
116  policy.priv_tomodify = tagbit;
117  policy.priv_set = 0;
118  ptaxi_add_policy(policy);
119 }
120
```

```
121 uint8_t ptaxi_base_policy_gettag(uint8_t tagbit, void *addr) {
122   uint64_t out;
123   uint64_t cmd = PTAXI_TAGCMD_PREFIX_GETTAG + log2bit(tagbit);
124   uint64_t in = *((uint64_t *) addr);
125   __asm__ volatile ("tagcmd %0,%2,%1" : "=r"(out) :"r"(in), "r"(cmd)
      );
126   return out;
127 }
128
129 // tagval is a boolean. True = set/False = clear
130 void ptaxi_base_policy_settag(uint8_t tagbit, void *addr, uint8_t
      tagval) {
131   uint64_t cmd;
132   if (tagval > 0) {
133     cmd = PTAXI_TAGCMD_PREFIX_SETTAG + log2bit(tagbit);
134   } else {
135     cmd = PTAXI_TAGCMD_PREFIX_CLEARTAG + log2bit(tagbit);
136   }
137   uint64_t *raddr = (uint64_t *) addr;
138   uint64_t in = *raddr;
139   uint64_t out = 0;
140   __asm__ volatile ("tagcmd %0,%2,%1" : "=r"(out) :"r"(in), "r"(cmd)
      );
141   *raddr = out;
142 }
143
144 void ptaxi_base_policy_settag_multi(uint8_t tagbit, void *addr,
      size_t size, uint8_t tagval) {
145   size_t block = size/4;
146   uint64_t* pos = (uint64_t *) addr;
147   size_t i;
148   for (i = 0; i < block; i++) {
149     ptaxi_base_policy_settag(tagbit, addr, tagval);
150     pos++;
151   }
152 }
153
154 // privval is a boolean. True = set/False = clear
155 void ptaxi_base_policy_setpriv(uint8_t tagbit, uint8_t privval) {
156   uint8_t cmd;
157   if (privval > 0) {
158     cmd = PTAXI_TAGCMD_PREFIX_SETPRIV + log2bit(tagbit);
159   } else {
160     cmd = PTAXI_TAGCMD_PREFIX_CLEARPRIV + log2bit(tagbit);
161   }
162   ptaxi_tag_command(cmd);
163 }
164
165 #endif
```

Listing B.1: `ptaxi_base_policy.h`

B.1.2 Return Address Protection

```
#ifndef _PTAXI_POLICY_RETURN_ADDRESS_H
#define _PTAXI_POLICY_RETURN_ADDRESS_H

#include <stdlib.h>

#include "ptaxi_common.h"
#include "ptaxi_user.h"
#include "ptaxi_base_policy.h"

void ptaxi_policy_return_address(uint8_t tagbit) {
  struct ptaxi_policy_t default_policy, policy;
  memset(&default_policy, 0, sizeof(default_policy));
  policy = default_policy;
  policy.insn_type = PTAXI_INSN_TYPE_JAL;
  policy.tag_out_tomodify = tagbit;
  policy.tag_out_set = tagbit;
  ptaxi_add_policy(policy);

  policy = default_policy;
  policy.insn_type = PTAXI_INSN_TYPE_RETURN;
  policy.tag_arg1_mask = tagbit;
  policy.tag_arg1_match = 0;
  policy.action |= PTAXI_ACTION_BLOCK;
  policy.ignore_count = 4;
  ptaxi_add_policy(policy);

  ptaxi_base_policy_propatgate_by_type(tagbit,
   PTAXI_INSN_TYPE_STORE64, 1, 0);
  ptaxi_base_policy_propatgate_by_type(tagbit,
   PTAXI_INSN_TYPE_LOAD64, 1, 0);
  ptaxi_base_policy_propatgate_by_type(tagbit, PTAXI_INSN_TYPE_COPY,
    1, 0);
  ptaxi_base_policy_clear_by_type(tagbit, PTAXI_INSN_TYPE_STORE);
  ptaxi_base_policy_clear_by_type(tagbit, PTAXI_INSN_TYPE_LOAD);
  ptaxi_base_policy_clear_by_type(tagbit, PTAXI_INSN_TYPE_OP);
  ptaxi_base_policy_clear_by_type(tagbit, PTAXI_INSN_TYPE_OPIMM);
}

#endif
```

Listing B.2: `ptaxi_policy_return_address.h`

B.1.3 Memory Compartmentalization & Taint Tracking (Privilege)

```
#ifndef _PTAXI_POLICY_PRIVILEGE_H
#define _PTAXI_POLICY_PRIVILEGE_H

```

```
4  void ptaxi_policy_privilege_init(uint8_t tagbit) {
5    ptaxi_base_policy_create_settag(tagbit);
6    ptaxi_base_policy_create_gettag(tagbit);
7    ptaxi_base_policy_create_cleartag(tagbit);
8    ptaxi_base_policy_create_setpriv(tagbit);
9    ptaxi_base_policy_create_clearpriv(tagbit);
10   ptaxi_base_policy_propatgate_by_type(tagbit,
       PTAXI_INSN_TYPE_STORE64, 1, 0);
11   ptaxi_base_policy_propatgate_by_type(tagbit,
       PTAXI_INSN_TYPE_LOAD64, 1, 0);
12   ptaxi_base_policy_propatgate_by_type(tagbit, PTAXI_INSN_TYPE_COPY,
       1, 0);
13 }
14
15 void ptaxi_policy_privilege_protect_tag_from_nonprivilege(uint8_t
       tagbit) {
16
17 }
18
19 void ptaxi_policy_privilege_protect_data(uint8_t tagbit) {
20   struct ptaxi_policy_t default_policy, policy;
21   memset(&default_policy, 0, sizeof(default_policy));
22   default_policy.priv_mask = tagbit;
23   default_policy.priv_match = 0;
24   default_policy.action = PTAXI_ACTION_BLOCK;
25
26   policy = default_policy;
27   policy.insn_type = PTAXI_INSN_TYPE_LOAD;
28   policy.tag_arg1_mask = tagbit;
29   policy.tag_arg1_match = tagbit;
30   ptaxi_add_policy(policy);
31
32   policy = default_policy;
33   policy.insn_type = PTAXI_INSN_TYPE_LOAD64;
34   policy.tag_arg1_mask = tagbit;
35   policy.tag_arg1_match = tagbit;
36   ptaxi_add_policy(policy);
37
38   policy = default_policy;
39   policy.insn_type = PTAXI_INSN_TYPE_STORE;
40   policy.tag_out_mask = tagbit;
41   policy.tag_out_match = tagbit;
42   ptaxi_add_policy(policy);
43
44   policy = default_policy;
45   policy.insn_type = PTAXI_INSN_TYPE_STORE64;
46   policy.tag_out_mask = tagbit;
47   policy.tag_out_match = tagbit;
48   ptaxi_add_policy(policy);
49 }
50
51 void ptaxi_policy_privilege_enter(uint8_t tagbit) {
52   ptaxi_base_policy_setpriv(tagbit, 1);
```

```
}

void ptaxi_policy_privilege_leave(uint8_t tagbit) {
  ptaxi_base_policy_setpriv(tagbit, 0);

}
#endif
```

Listing B.3: `ptaxi_policy_privilege.h`

B.1.4 Stack Garbage Collection

```
#ifndef _PTAXI_POLICY_GC_H
#define _PTAXI_POLICY_GC_H

#include <stdlib.h>

#include "ptaxi_common.h"
#include "ptaxi_user.h"

void ptaxi_policy_gc() {
  struct ptaxi_policy_t policy;
  memset(&policy, 0, sizeof(policy));
  policy.insn_type = PTAXI_INSN_TYPE_RETURN;
  policy.action = PTAXI_ACTION_GC;
  ptaxi_add_policy(policy);
}

#endif
```

Listing B.4: `ptaxi_policy_gc.h`

B.1.5 Call Debugging

```
#ifndef _PTAXI_POLICY_DEBUG_CALL_H
#define _PTAXI_POLICY_DEBUG_CALL_H

#include <stdlib.h>

#include "ptaxi_common.h"
#include "ptaxi_user.h"

void ptaxi_policy_debug_call() {
  struct ptaxi_policy_t default_policy, policy;
  memset(&default_policy, 0, sizeof(default_policy));

  policy = default_policy;
  policy.insn_type = PTAXI_INSN_TYPE_JAL;
  policy.action = PTAXI_ACTION_DEBUG_LINE;
  ptaxi_add_policy(policy);

```

```
  policy = default_policy;
  policy.insn_type = PTAXI_INSN_TYPE_JALR;
  policy.action = PTAXI_ACTION_DEBUG_LINE;
  ptaxi_add_policy(policy);

  policy = default_policy;
  policy.insn_type = PTAXI_INSN_TYPE_RETURN;
  policy.action = PTAXI_ACTION_DEBUG_LINE;
  ptaxi_add_policy(policy);
}

#endif
```

Listing B.5: `ptaxi_policy_debug_call.h`

B.2 Test Cases

B.2.1 Return Address Protection

```
#include <stdio.h>
#include "ptaxi.h"

#define TAG_RETURNADDRESS 1

void __attribute__ ((constructor)) ptaxi_app_policy() {
  ptaxi_policy_return_address(TAG_RETURNADDRESS);
  ptaxi_enforce_policy();
}

int i;
int limit = 20;
int g(int x) {
  return x * 27 % 13; // To prevent compiler optimizations
}

int f() {
  uint64_t a[2];
  a[i] = 0xDEADBEEF;
  for (i = 1; i < limit; i++) {
    a[i] = a[i - 1] + g(i);
  }
  return 7;
}
int main(int argc, char** argv) {
  if (argc > 1) {
    printf("Should pass instead of fail.\n");
    limit = 2;
  }
  f();
  return 0;
```

```
}
```

Listing B.6: `test_return_address.c`

B.2.2 Get and Set Tags

```
#include <stdio.h>
#include <unistd.h>

#include "ptaxi.h"

#define TAGBIT 1

void __attribute__ ((constructor)) ptaxi_app_policy() {
  ptaxi_policy_privilege_init(TAGBIT);
  ptaxi_enforce_policy();
}

int main(int argc, char** argv) {
  uint64_t a = 5;
  printf("A is at %p\n", &a);
  printf ("A = %lu, TAG(A) = %d (should be 0)\n", a, (int)
    ptaxi_base_policy_gettag(TAGBIT, &a));
  ptaxi_base_policy_settag(TAGBIT, &a, 1);
  printf ("A = %lu, TAG(A) = %d (should be 1)\n", a, (int)
    ptaxi_base_policy_gettag(TAGBIT, &a));
  ptaxi_base_policy_settag(TAGBIT, &a, 1);
  printf ("A = %lu, TAG(A) = %d (should be 1)\n", a, (int)
    ptaxi_base_policy_gettag(TAGBIT, &a));
  ptaxi_base_policy_settag(TAGBIT, &a, 0);
  printf ("A = %lu, TAG(A) = %d (should be 0)\n", a, (int)
    ptaxi_base_policy_gettag(TAGBIT, &a));
  return 0;
}
```

Listing B.7: `test_getsettag.c`

B.2.3 Malloc with Memory Compartmentalization

```
#include <stdio.h>
#include <unistd.h>

#include "ptaxi.h"

#define MALLOC_TAGBIT 4

typedef struct malloc_ll {
  size_t size;
  struct malloc_ll *back;
  struct malloc_ll *next;
} malloc_ll;
```

```
13
14 malloc_ll *malloc_root;
15
16 void __attribute__ ((constructor)) ptaxi_app_policy() {
17   ptaxi_policy_privilege_protect_data(MALLOC_TAGBIT);
18   ptaxi_policy_privilege_init(MALLOC_TAGBIT);
19   ptaxi_enforce_policy();
20 }
21
22 void *pmalloc_internal(size_t size) {
23   malloc_ll *node = sbrk(sizeof(malloc_ll) + size);
24   node->size = size;
25   node->back = NULL;
26   node->next = malloc_root;
27   if (malloc_root != NULL) {
28     malloc_root->back = node;
29   }
30   ptaxi_base_policy_settag_multi(MALLOC_TAGBIT, node, sizeof(
     malloc_ll), 1);
31   malloc_root = node;
32   void *res = ((void *) node) + sizeof(malloc_ll);
33   return res;
34 }
35
36 void *pmalloc(size_t size) {
37   ptaxi_policy_privilege_enter(MALLOC_TAGBIT);
38   void *res = pmalloc_internal(size);
39   ptaxi_policy_privilege_leave(MALLOC_TAGBIT);
40   return res;
41 }
42
43 void read_size(char *a) {
44   malloc_ll *m = (malloc_ll *) (a - sizeof(malloc_ll));
45   size_t size = m->size;    // Should trap here!
46   printf("Malloc Size = %d \n", size);
47 }
48
49 int main(int argc, char** argv) {
50
51   char *a = (char *) pmalloc(48);
52   a[0] = 'H';
53   a[1] = 'E';
54   a[2] = 'L';
55   a[3] = 'L';
56   a[4] = 'O';
57   a[5] = '\0';
58
59   printf("String = %s\n", a);
60   if (argc > 1) {
61     // Should pass instead of fail.
62   } else {
63     read_size(a);
64   }
```

```
65   return 0;
66 }
```

Listing B.8: `test_simple_malloc.c`

B.2.4 Basic Taint Tracking

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  #include "ptaxi.h"
5
6  #define TAINT_TAGBIT 1
7
8  void __attribute__ ((constructor)) ptaxi_app_policy() {
9    ptaxi_policy_privilege_init(TAINT_TAGBIT);
10   ptaxi_base_policy_propatgate_by_type(TAINT_TAGBIT,
       PTAXI_INSN_TYPE_OP, 1, 1);
11   ptaxi_base_policy_propatgate_by_type(TAINT_TAGBIT,
       PTAXI_INSN_TYPE_OPIMM, 1, 0);
12   ptaxi_enforce_policy();
13 }
14
15 uint64_t get_unfiltered_input() {
16   uint64_t input = 42;
17   ptaxi_base_policy_settag(TAINT_TAGBIT, (void *) (&input), 1);
18   printf("DEBUGE: %d\n", (int) ptaxi_base_policy_gettag(TAINT_TAGBIT
       , (void *) (&input)));
19   return input;
20 }
21
22 int main(int argc, char** argv) {
23   uint64_t a = get_unfiltered_input();
24   uint64_t b = 4;
25   uint64_t c = b * 20;
26   uint64_t d = a + 5;
27
28   int s1 = ptaxi_base_policy_gettag(TAINT_TAGBIT, (void *) (&c));
29   int s2 = ptaxi_base_policy_gettag(TAINT_TAGBIT, (void *) (&d));
30   printf("TAG(C) = %d (should be 0), TAG(D) = %d (should be 1)\n",
       s1, s2);
31
32   return 0;
33 }
```

Listing B.9: `test_taint_tracking.c`

B.2.5 Stack Garbage Collection

```
1  #include <stdio.h>
2  #include "ptaxi.h"
```

```
3
4  uint64_t *gsecretnumberptr = NULL;
5
6  int i;
7
8  int g(int x) {
9    return x * 2 + 9;
10 }
11
12 int f() {
13   ptaxi_tag_command(123);
14   uint64_t secretnumber = 0xDEADBEEF;
15   gsecretnumberptr = &secretnumber;
16   int i, s = 0;
17   for (i = 0; i < 20; i++) {
18     s += g(i) + secretnumber;
19   }
20   ptaxi_tag_command(456);
21   return s;
22 }
23
24 int main(int argc, char** argv) {
25   if (!(argc > 1)) {
26     ptaxi_policy_gc();
27     ptaxi_enforce_policy();
28   } else {
29     printf("Should show deadbeef\n");
30   }
31   ptaxi_tag_command(7);
32   f();
33   ptaxi_tag_command(8);
34   printf("Secret = %x at %p\n", *gsecretnumberptr, (void *)
      gsecretnumberptr);
35
36   return 0;
37 }
```

Listing B.10: `test_gc.c`

Bibliography

[1] I. Evans, "Analysis of defenses against code reuse attacks on modern and new architectures," PhD thesis, Department of Electrical Engineering et al., 2015. [Online]. Available: http://people.csail.mit.edu/hes/ROP/Publications/Isaac-thesis.pdf (visited on 05/06/2016).

[2] S. Fingeret, "Defeating code reuse attacks with minimal tagged architecture," PhD thesis, Massachusetts Institute of Technology, 2015. [Online]. Available: http://people.csail.mit.edu/hes/ROP/Publications/sam-thesis.pdf (visited on 04/20/2016).

[3] J. A. González, "Taxi: Defeating code reuse attacks with tagged memory," PhD thesis, Massachusetts Institute of Technology, 2015. [Online]. Available: http://people.csail.mit.edu/hes/ROP/Publications/Julian-thesis.pdf (visited on 04/20/2016).

[4] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight Jr., B. C. Pierce, and A. DeHon, "Architectural support for software-defined metadata processing," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15, New York, NY, USA: ACM, 2015, pp. 487–502, ISBN: 978-1-4503-2835-7. DOI: 10.1145/2694344.2694383.

[5] J. P. Anderson, "Computer security technology planning study. volume 2," Oct. 1972. [Online]. Available: http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=AD0772806 (visited on 05/01/2015).

[6] B. Martin, M. Brown, A. Paller, D. Kirby, and S. Christey, "2011 CWE/SANS top 25 most dangerous software errors," *Common Weakness Enumeration*, vol. 7515, 2011. [Online]. Available: http://cwe.mitre.org/top25/archive/2010/2010_cwe_sans_top25.pdf (visited on 05/01/2015).

[7] Aleph One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, Aug. 11, 1996. [Online]. Available: http://phrack.org/issues/49/14.html (visited on 04/20/2016).

[8] S. Friedl. (2004). Intel x86 function-call conventions - assembly view, Unixwiz.net, [Online]. Available: http://unixwiz.net/techtips/win32-callconv-asm.html (visited on 04/27/2016).

[9] Microsoft. (Dec. 2010). Windows ISV software security defenses, [Online]. Available: https://msdn.microsoft.com/en-us/library/bb430720.aspx (visited on 05/01/2015).

[10] Oracle. (2013). Configuring and using kernel security mechanisms, [Online]. Available: https://docs.oracle.com/cd/E37670_01/E36387/html/ol_kernel_sec.html (visited on 05/01/2015).

[11] Apple. (2014). OS x - security, Apple, [Online]. Available: https://www.apple.com/osx/what-is/security/ (visited on 04/27/2016).

[12] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, ser. SSYM'98, Berkeley, CA, USA: USENIX Association, 1998, pp. 5–5. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267549.1267554.

[13] Free Software Foundation. (Jan. 30, 2016). GCC 4.1 release series changes, new features, and fixes - GNU project - free software foundation (FSF), [Online]. Available: https://gcc.gnu.org/gcc-4.1/changes.html (visited on 04/26/2016).

[14] G. Richarte, "Four different tricks to bypass stackshield and stackguard protection," *Core Security Technologies*, 2002. [Online]. Available: http://www.coresecurity.com/system/files/StackguardPaper.pdf (visited on 04/26/2016).

[15] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11, New York, NY, USA: ACM, 2011, pp. 30–40, ISBN: 978-1-4503-0564-8. DOI: 10.1145/1966913.1966919.

[16] R. Wojtczuk, "The advanced return-into-lib (c) exploits: PaX case study," *Phrack magazine*, vol. 11, no. 58, p. 4, Dec. 28, 2001. [Online]. Available: http://phrack.org/issues/58/4.html.

[17] H. 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*, ser. CCS '07, New York, NY, USA: ACM, 2007, pp. 552–561, ISBN: 978-1-59593-703-2. DOI: 10.1145/1315245.1315313.

[18] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10, New York, NY, USA: ACM, 2010, pp. 559–572, ISBN: 978-1-4503-0245-6. DOI: 10.1145/1866307.1866370.

[19] L. Davi, A.-R. Sadeghi, and M. 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*, ser. ASIACCS '11, New York, NY, USA: ACM, 2011, pp. 40–51, ISBN: 978-1-4503-0564-8. DOI: 10.1145/1966913.1966920.

[20] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13, Berkeley, CA, USA: USENIX Association, 2013, pp. 447–462, ISBN: 978-1-931971-03-4. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534805.

[21] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "Ropecker: A generic and practical approach for defending against ROP attacks," Internet Society, 2014, ISBN: 978-1-891562-35-8. DOI: 10.14722/ndss.2014.23156.

[22] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14, Berkeley, CA, USA: USENIX Association, 2014, pp. 385–399, ISBN: 978-1-931971-15-7. [Online]. Available: http://dl.acm.org/citation.cfm?id=2671225.2671250.

[23] F. Schuster, T. Tendyck, J. Pewny, A. MaaSS, M. Steegmanns, M. Contag, and T. Holz, "Evaluating the effectiveness of current anti-ROP defenses," in *Research in Attacks, Intrusions and Defenses*, A. Stavrou, H. Bos, and G. Portokalidis, Eds., vol. 8688, Cham: Springer International Publishing, 2014, pp. 88–108, ISBN: 978-3-319-11378-4 978-3-319-11379-1. [Online]. Available: http://link.springer.com/10.1007/978-3-319-11379-1_5 (visited on 04/20/2016).

[24] PaX Team. (2003). PaX address space layout randomization (ASLR), [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt (visited on 05/01/2015).

[25] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, Dec. 2006, pp. 339–348. DOI: 10.1109/ACSAC.2006.9.

[26] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "Ilr: Where'd my gadgets go?" In *2012 IEEE Symposium on Security and Privacy*, May 2012, pp. 571–585. DOI: 10.1109/SP.2012.39.

[27] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "ASLR-guard: Stopping address space leakage for code reuse attacks," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, New York, NY, USA: ACM, 2015, pp. 280–291, ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813694.

[28] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, New York, NY, USA: ACM, 2015, pp. 268–279, ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813691.

[29] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04, New York, NY, USA: ACM, 2004, pp. 298–307, ISBN: 978-1-58113-961-7. DOI: 10.1145/1030083.1030124.

[30] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *2013 IEEE Symposium on Security and Privacy (SP)*, May 2013, pp. 574–588. DOI: 10.1109/SP.2013.45.

[31] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *2014 IEEE Symposium on Security and Privacy*, May 2014, pp. 227–242. DOI: 10.1109/SP.2014.22.

[32] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14, New York, NY, USA: ACM, 2014, pp. 54–65, ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660309.

[33] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c," in *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '02, Berkeley, CA, USA: USENIX Association, 2002, pp. 275–288, ISBN: 978-1-880446-00-3. [Online]. Available: http://dl.acm.org/citation.cfm?id=647057.713871.

[34] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems*, vol. 27, no. 3, pp. 477–526, May 2005, ISSN: 0164-0925. DOI: 10.1145/1065887.1065892. (visited on 04/20/2016).

[35] N. D. Matsakis and F. S. Klock II, "The rust language," in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT '14, New York, NY, USA: ACM, 2014, pp. 103–104, ISBN: 978-1-4503-3217-0. DOI: 10.1145/2663171.2663188.

[36] Rust Project. (2016). The rust programming language, [Online]. Available: https://doc.rust-lang.org/book/ (visited on 05/06/2016).

[37] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: Architectural support for spatial safety of the c programming language," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII, New York,

NY, USA: ACM, 2008, pp. 103–114, ISBN: 978-1-59593-958-6. DOI: 10.1145/1346281.1346295.

[38] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, New York, NY, USA: ACM, 2009, pp. 245–258, ISBN: 978-1-60558-392-1. DOI: 10.1145/1542476.1542504.

[39] P. Akritidis, M. Costa, M. Castro, and S. 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*, ser. SSYM'09, Berkeley, CA, USA: USENIX Association, 2009, pp. 51–66. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855768.1855772.

[40] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14, Berkeley, CA, USA: USENIX Association, 2014, pp. 147–163, ISBN: 978-1-931971-16-4. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685061.

[41] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the point(er): On the effectiveness of code pointer integrity," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 781–796. DOI: 10.1109/SP.2015.53.

[42] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, and D. Song, "Poster: Getting the point (er): On the feasibility of attacks on code-pointer integrity," in *IEEE Symposium on Security and Privacy*, 2015. [Online]. Available: http://www.ieee-security.org/TC/SP2015/posters/paper_48.pdf.

[43] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05, New York, NY, USA: ACM, 2005, pp. 340–353, ISBN: 978-1-59593-226-6. DOI: 10.1145/1102120.1102165.

[44] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *2013 IEEE Symposium on Security and Privacy (SP)*, May 2013, pp. 559–573. DOI: 10.1109/SP.2013.44.

[45] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," presented at the Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13), 2013, pp. 337–352, ISBN: 978-1-931971-03-4. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang.

[46] ——, "Control flow and code integrity for COTS binaries: An effective defense against real-world ROP attacks," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC 2015, New York, NY, USA: ACM, 2015, pp. 91–100, ISBN: 978-1-4503-3682-6. DOI: 10.1145/2818000.2818016.

[47] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "Ccfi: Cryptographically enforced control flow integrity," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, New York, NY, USA: ACM, 2015, pp. 941–951, ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813676.

[48] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity," Internet Society, 2015, ISBN: 978-1-891562-38-9. DOI: 10.14722/ndss.2015.23271.

[49] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14, Berkeley, CA, USA: USENIX Association, 2014, pp. 401–416, ISBN: 978-1-931971-15-7. [Online]. Available: http://dl.acm.org/citation.cfm?id=2671225.2671251.

[50] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *2014 IEEE Symposium on Security and Privacy (SP)*, May 2014, pp. 575–589. DOI: 10.1109/SP.2014.43.

[51] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 745–762. DOI: 10.1109/SP.2015.51.

[52] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, New York, NY, USA: ACM, 2015, pp. 901–913, ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813646.

[53] E. A. Feustel, "On the advantages of tagged architecture," *IEEE Transactions on Computers*, vol. C-22, no. 7, pp. 644–656, Jul. 1973, ISSN: 0018-9340. DOI: 10.1109/TC.1973.5009130.

[54] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14, Piscataway, NJ, USA: IEEE Press, 2014, pp. 457–468, ISBN: 978-1-4799-4394-4. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665740.

[55] D. Chisnall, C. Rothwell, R. N. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann, "Beyond the PDP-11: Architectural support for a memory-safe c abstract machine," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15, New York, NY, USA: ACM, 2015, pp. 117–130, ISBN: 978-1-4503-2835-7. DOI: 10.1145/2694344.2694367.

[56] A. A. d. Amorim, M. Dénès, N. Giannarakis, C. Hritcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach, "Micro-policies: Formally verified, tag-based security monitors," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 813–830. DOI: 10.1109/SP.2015.55.

[57] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, "The RISC-v instruction set manual, volume i: User-level ISA, version 2.0," EECS Department, University of California, Berkeley, May 6, 2014. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html.

[58] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovi, "The RISC-v instruction set manual volume II: Privileged architecture version 1.7," EECS Department, University of California, Berkeley, May 9, 2015. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-49.html.

[59] Q. Nguyen. (Mar. 21, 2013). The RISC-v linux user's manual, [Online]. Available: https://www.ocf.berkeley.edu/~qmn/linux/riscv.html (visited on 05/06/2016).

[60] A. Waterman and Y. Lee. (Jun. 19, 2011). RISC-v ISA simulator, RISC-V Foundation, [Online]. Available: http://riscv.org/software-tools/risc-v-isa-simulator/ (visited on 05/06/2016).

[61] University of California. (2013). Riscv/riscv-pk, GitHub, [Online]. Available: https://github.com/riscv/riscv-pk (visited on 05/06/2016).

[62] R. Cielak. (Apr. 2, 2013). Dynamic linker tricks: Using LD_preload to cheat, inject features and investigate programs, Rafa Cielak's blog, [Online]. Available: https://rafalcieslak.wordpress.com/2013/04/02/dynamic-linker-tricks-using-ld_preload-to-cheat-inject-features-and-investigate-programs/ (visited on 05/12/2016).

[63] P. Saxena, R. Sekar, and V. Puranik, "Efficient fine-grained binary instrumentationwith applications to taint-tracking," in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '08, New York, NY, USA: ACM, 2008, pp. 74–83, ISBN: 978-1-59593-978-4. DOI: 10.1145/1356058.1356069.

[64] M.-K. Yoon, N. Salajegheh, Y. Chen, and M. Christodorescu, "Pift: Predictive information-flow tracking," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16, New York, NY, USA: ACM, 2016, pp. 713–725, ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872403.

[65] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, and P. Saxena, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy*, to be published, 2016. [Online]. Available: https://www.comp.nus.edu.sg/~shweta24/publications/dop_oakland16.pdf.

[66] A. Reece. (May 2, 2013). Introduction to format string exploits, Code Arcana, [Online]. Available: http://codearcana.com/posts/2013/05/02/introduction-to-format-string-exploits.html (visited on 05/12/2016).