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

1 Introduction ................................................. 19

2 Background .................................................. 21

   2.1 Code-Injection Attacks .................................. 21

      2.1.1 Example ............................................. 21

      2.1.2 Defenses ............................................ 23

         2.1.2.1 Data Execution Prevention (DEP) ............ 23

         2.1.2.2 StackGuard ..................................... 24

   2.2 Code-Reuse Attack ...................................... 25

      2.2.1 Return-To-Libc Attack ............................. 26

      2.2.2 Return-Oriented Programming ..................... 26

3 Return-Oriented Programming ............................... 27

   3.1 Variants of ROP Attacks ................................. 28

      3.1.1 ROP Using pop and jmp Instructions ............. 28

      3.1.2 Jump-Oriented Programming ....................... 28

   3.2 Detection-Based Defenses ............................... 28

      3.2.1 ROPdefender ....................................... 28

      3.2.2 kBouncer .......................................... 29

      3.2.3 ROPecker .......................................... 29

      3.2.4 Circumvention ..................................... 29

   3.3 Address Space Layout Randomization (ASLR) ............ 30

      3.3.1 Variants of ASLR .................................. 31
<p>| 3.3.1.1 Address Space Layout Permutation (ASLP) | 31 |
| 3.3.1.2 Instruction Location Randomization (ILR) | 31 |
| 3.3.1.3 ASLR-Guard | 31 |
| 3.3.1.4 Timely Address Space Randomization (TASR) | 32 |
| 3.3.2 Circumvention | 32 |
| 3.3.2.1 Derandomization Attack | 32 |
| 3.3.2.2 Just-In-Time Code Reuse | 33 |
| 3.3.2.3 Blind ROP | 33 |
| 3.3.2.4 Side Channel Attacks | 33 |
| 3.4 Annotated Language | 34 |
| 3.4.1 Cyclone | 34 |
| 3.4.2 CCured | 34 |
| 3.4.3 Rust | 35 |
| 3.5 Bound-Based Defenses | 35 |
| 3.5.1 HardBound | 35 |
| 3.5.2 SoftBound | 36 |
| 3.5.3 Baggy Bound | 36 |
| 3.6 Code-Pointer Integrity (CPI) | 36 |
| 3.6.1 Circumvention | 37 |
| 3.7 Control-Flow Integrity (CFI) | 37 |
| 3.7.1 Variants of CFI | 38 |
| 3.7.1.1 Compact Control Flow Integrity and Randomization (CCFIR) | 38 |
| 3.7.1.2 Control Flow and Code Integrity (CFCI) | 38 |
| 3.7.1.3 Cryptographically-Enforced Control Flow Integrity (CCFI) | 38 |
| 3.7.1.4 Opaque Control-Flow Integrity (O-CFI) | 38 |
| 3.7.2 Circumvention | 39 |
| 3.7.2.1 Coarse-Grained CFI | 39 |
| 3.7.2.2 Problem with Static Analysis | 39 |</p>
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.7.2.3</td>
<td>Counterfeit Object-Oriented Programming (COOP)</td>
<td>39</td>
</tr>
<tr>
<td>3.7.2.4</td>
<td>Control Jujutsu</td>
<td>39</td>
</tr>
<tr>
<td>4</td>
<td>Tagged Architecture</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>Capability Hardware Enhanced RISC Instructions (CHERI)</td>
<td>41</td>
</tr>
<tr>
<td>4.2</td>
<td>Programmable Unit for Metadata Processing (PUMP)</td>
<td>42</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Policies</td>
<td>43</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Implementation</td>
<td>43</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Policy Correctness</td>
<td>44</td>
</tr>
<tr>
<td>5</td>
<td>RISC-V</td>
<td>45</td>
</tr>
<tr>
<td>5.1</td>
<td>Instruction Set</td>
<td>45</td>
</tr>
<tr>
<td>5.2</td>
<td>Registers</td>
<td>47</td>
</tr>
<tr>
<td>5.3</td>
<td>Spike</td>
<td>47</td>
</tr>
<tr>
<td>5.4</td>
<td>Proxy Kernel (PK)</td>
<td>47</td>
</tr>
<tr>
<td>6</td>
<td>TAXI</td>
<td>49</td>
</tr>
<tr>
<td>6.1</td>
<td>Tagged Architecture</td>
<td>49</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Tag Unit</td>
<td>50</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Tag Cache</td>
<td>50</td>
</tr>
<tr>
<td>6.2</td>
<td>Policies</td>
<td>50</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Return Address Protection</td>
<td>50</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Linearity of Return Address</td>
<td>51</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Data Blacklisting</td>
<td>51</td>
</tr>
<tr>
<td>6.3</td>
<td>Performance</td>
<td>52</td>
</tr>
<tr>
<td>7</td>
<td>P-TAXI</td>
<td>53</td>
</tr>
<tr>
<td>7.1</td>
<td>Contributions</td>
<td>53</td>
</tr>
<tr>
<td>7.2</td>
<td>Threat Model</td>
<td>54</td>
</tr>
<tr>
<td>7.3</td>
<td>Design</td>
<td>54</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Policy</td>
<td>55</td>
</tr>
<tr>
<td>7.3.1.1</td>
<td>Filter</td>
<td>55</td>
</tr>
</tbody>
</table>
# A P-TAXI Source Code

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 Policy Definition ([ptaxi_common.h])</td>
<td>75</td>
</tr>
<tr>
<td>A.2 P-TAXI Simulator</td>
<td>77</td>
</tr>
<tr>
<td>A.2.1 Header File ([ptaxisim.h])</td>
<td>77</td>
</tr>
<tr>
<td>A.2.2 Source File ([ptaxisim.cc])</td>
<td>78</td>
</tr>
<tr>
<td>A.3 User-level Libraries</td>
<td>90</td>
</tr>
<tr>
<td>A.3.1 Basic Interface ([ptaxi_user.h])</td>
<td>90</td>
</tr>
<tr>
<td>A.3.2 Header File for Inclusion by User-Level Applications ([ptaxi.h])</td>
<td>91</td>
</tr>
</tbody>
</table>

# B Policy Source Code and Test Cases

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1 Policies</td>
<td>93</td>
</tr>
<tr>
<td>B.1.1 Base Policies</td>
<td>93</td>
</tr>
<tr>
<td>B.1.2 Return Address Protection</td>
<td>97</td>
</tr>
<tr>
<td>B.1.3 Memory Compartmentalization &amp; Taint Tracking (Privilege)</td>
<td>97</td>
</tr>
<tr>
<td>B.1.4 Stack Garbage Collection</td>
<td>99</td>
</tr>
<tr>
<td>B.1.5 Call Debugging</td>
<td>99</td>
</tr>
<tr>
<td>B.2 Test Cases</td>
<td>100</td>
</tr>
<tr>
<td>B.2.1 Return Address Protection</td>
<td>100</td>
</tr>
<tr>
<td>B.2.2 Get and Set Tags</td>
<td>101</td>
</tr>
<tr>
<td>B.2.3 Malloc with Memory Compartmentalization</td>
<td>101</td>
</tr>
<tr>
<td>B.2.4 Basic Taint Tracking</td>
<td>103</td>
</tr>
<tr>
<td>B.2.5 Stack Garbage Collection</td>
<td>103</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Stack after the execution of line 4 with input “ABCD”</td>
<td>22</td>
</tr>
<tr>
<td>2-2</td>
<td>Stack after the execution of line 4 with input “AAAAAAAAAAAAAAAAAAAA”</td>
<td>23</td>
</tr>
<tr>
<td>2-3</td>
<td>Stack in the same condition as in figure 2-1 but with StackGuard enabled.</td>
<td>25</td>
</tr>
<tr>
<td>2-4</td>
<td>Stack after the execution of line 4 with input “AAAAAAAAAAA&lt;address to system&gt;”</td>
<td>26</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Numbers of general-purpose RISC-V instructions, categorized by extension. ........................................ 46
5.2 List of RISC-V opcodes used by general-purpose instructions. ....................................................... 46
7.1 List of fields available for each P-TAXI policy. ................................................................. 55
7.2 List of P-TAXI instruction types. ...................................................................................... 57
7.3 List of P-TAXI actions. ................................................................................................. 58
8.1 List of base sets of policies implemented in the P-TAXI user-level libraries. 64
8.2 List of P-TAXI policies used to implement the base sets of policies. .... 64
8.3 List of policies for return address protection. ................................................................. 65
8.4 List of policies for memory compartmentalization. ..................................................... 67
8.5 List of policies for taint tracking. .................................................................................. 68
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. ..................................................... 74
## List of Code Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example C program with a buffer overflow vulnerability.</td>
<td>22</td>
</tr>
<tr>
<td>7.1</td>
<td>Example C program that utilizes <code>ptaxi.h</code>.</td>
<td>61</td>
</tr>
<tr>
<td>7.2</td>
<td>Example C library that can be loaded into existing programs to enable P-TAXI via <code>LD_PRELOAD</code>.</td>
<td>61</td>
</tr>
<tr>
<td>8.1</td>
<td>Malloc implementation that utilizes P-TAXI policies to enable memory compartmentalization.</td>
<td>66</td>
</tr>
<tr>
<td>8.2</td>
<td>Example code utilizing taint tracking.</td>
<td>68</td>
</tr>
<tr>
<td>A.1</td>
<td><code>ptaxi_common.h</code></td>
<td>75</td>
</tr>
<tr>
<td>A.2</td>
<td><code>ptaxisim.h</code></td>
<td>77</td>
</tr>
<tr>
<td>A.3</td>
<td><code>ptaxisim.cc</code></td>
<td>78</td>
</tr>
<tr>
<td>A.4</td>
<td><code>ptaxi_user.h</code></td>
<td>90</td>
</tr>
<tr>
<td>A.5</td>
<td><code>ptaxi.h</code></td>
<td>91</td>
</tr>
<tr>
<td>B.1</td>
<td><code>ptaxi_base_policy.h</code></td>
<td>93</td>
</tr>
<tr>
<td>B.2</td>
<td><code>ptaxi_policy_return_address.h</code></td>
<td>97</td>
</tr>
<tr>
<td>B.3</td>
<td><code>ptaxi_policy_privilege.h</code></td>
<td>97</td>
</tr>
<tr>
<td>B.4</td>
<td><code>ptaxi_policy_gc.h</code></td>
<td>99</td>
</tr>
<tr>
<td>B.5</td>
<td><code>ptaxi_policy_debug_call.h</code></td>
<td>99</td>
</tr>
<tr>
<td>B.6</td>
<td><code>test_return_address.c</code></td>
<td>100</td>
</tr>
<tr>
<td>B.7</td>
<td><code>test_getsettag.c</code></td>
<td>101</td>
</tr>
<tr>
<td>B.8</td>
<td><code>test_simple_malloc.c</code></td>
<td>101</td>
</tr>
<tr>
<td>B.9</td>
<td><code>test_taint_tracking.c</code></td>
<td>103</td>
</tr>
<tr>
<td>B.10</td>
<td><code>test_gc.c</code></td>
<td>103</td>
</tr>
</tbody>
</table>
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.

```c
#include <stdio.h>
void f() {
    char name[6];
    gets(name);
    printf("Your name is %s\n", name);
}
void main() {
    f();
    return 0;
}
```

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

![Stack Diagram](#)

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 “AAAAAAAAAAAAAAAA”, 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 “AAAAAAAAAAAAAAAAAA”.

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) \cite{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 \texttt{-fno-stack-protector} when compiling the program \cite{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 \cite{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 \cite{14}.
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-libc attack to execute an arbitrary command by forcing a call to the `system` function in the standard C library [16].

![Stack diagram](image)

Figure 2-4: Stack after the execution of line 4 with input “AAAAAAAAAAA<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 \texttt{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 \cite{19}.

\subsection*{3.2.2 kBouncer}

kBouncer is an implementation of two ROP defense approaches. First, kBouncer
checks that for each execution of a \texttt{ret} instruction, the return address that appears
on the stack points back to valid call sites, which are locations that occur immediately
after \texttt{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 \cite{20}.

\subsection*{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 \cite{21}.

\subsection*{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
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
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 $\log_2(\text{size})$ 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 \texttt{ret} and \texttt{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
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.

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.

In an attempt to retain the compatibility with C programs, Chisnall et al. 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 \cite{riscv_vol1} and volume II \cite{riscv_vol2}.

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.

<table>
<thead>
<tr>
<th>Extension</th>
<th>32 bits</th>
<th>64 bits</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Integer</td>
<td>47</td>
<td>15</td>
<td>62</td>
</tr>
<tr>
<td>Standard Extension for Atomic (A)</td>
<td>11</td>
<td>11</td>
<td>22</td>
</tr>
<tr>
<td>Standard Extension for Multiplication and Division (M)</td>
<td>8</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>Standard Extension for Single-Precision Floating Point (F)</td>
<td>34</td>
<td>4</td>
<td>38</td>
</tr>
<tr>
<td>Standard Extension for Double-Precision Floating Point (D)</td>
<td>26</td>
<td>6</td>
<td>32</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>126</strong></td>
<td><strong>41</strong></td>
<td><strong>167</strong></td>
</tr>
</tbody>
</table>

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.

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

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 \texttt{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 \texttt{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.

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

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

Table 7.2: List of P-TAXI instruction types.
<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTINUE</td>
<td>Continue to apply next policies.</td>
</tr>
<tr>
<td>ALLOW</td>
<td>Allow the execution of the instruction and stop the policy processing for this instruction.</td>
</tr>
<tr>
<td>BLOCK</td>
<td>Block the execution of the instruction (cause trap) and stop the policy processing for this instruction.</td>
</tr>
<tr>
<td>GC</td>
<td>Do stack garbage collection and stop the policy processing for this instruction.</td>
</tr>
<tr>
<td>DEBUG_LINE</td>
<td>Show brief debug information.</td>
</tr>
<tr>
<td>DEBUG_DETAIL</td>
<td>Show detailed debug information.</td>
</tr>
<tr>
<td>GETTAG</td>
<td>Make TAGCMD returns the tag value of RS2 register (should use only with TAGCMD instruction).</td>
</tr>
</tbody>
</table>

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 \texttt{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 \texttt{GC} action is requested.

However, since the \texttt{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 \texttt{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.

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

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

<table>
<thead>
<tr>
<th>Name</th>
<th>#</th>
<th>Instruction Type</th>
<th>Additional conditions</th>
<th>Action</th>
<th>Output tag</th>
<th>Priv. Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROPAGATE</td>
<td>1</td>
<td>&lt;Specified&gt;</td>
<td>TAG(ARG1) = 1</td>
<td>-</td>
<td>Clear</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>&quot;&quot;</td>
<td>TAG(ARG1) = 0</td>
<td>-</td>
<td>Set</td>
<td>-</td>
</tr>
<tr>
<td>CLEAR</td>
<td>1</td>
<td>&quot;&quot;</td>
<td>N/A</td>
<td>-</td>
<td>Clear</td>
<td>-</td>
</tr>
<tr>
<td>SETTAG</td>
<td>1</td>
<td>TAGCMD</td>
<td>RS1 = &lt;Specified&gt;</td>
<td>-</td>
<td>Set</td>
<td>-</td>
</tr>
<tr>
<td>CLEARTAG</td>
<td>1</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>-</td>
<td>Clear</td>
<td>-</td>
</tr>
<tr>
<td>GETTAG</td>
<td>1</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>GETTAG</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SETPRIV</td>
<td>1</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>-</td>
<td>-</td>
<td>Set</td>
</tr>
<tr>
<td>CLEARPRIV</td>
<td>1</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>-</td>
<td>-</td>
<td>Clear</td>
</tr>
</tbody>
</table>

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.

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

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.

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.
<table>
<thead>
<tr>
<th>#</th>
<th>Instruction Type</th>
<th>Additional conditions</th>
<th>Action</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LOAD, LOAD64, STORE, STORE64</td>
<td>TAG(ARG1) = 1, PRIV = 0</td>
<td>BLOCK</td>
<td>Block non-privilege code from loading or overwriting sensitive area.</td>
</tr>
<tr>
<td>2</td>
<td>LOAD64, STORE64, COPY</td>
<td>&lt;Base Policy: PROPAGATE&gt;</td>
<td>Propagate tags when they are copied.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>TAGCMD</td>
<td>&lt;Base Policy: SETTAG&gt;</td>
<td>Create SETTAG command.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>——”—”</td>
<td>&lt;Base Policy: CLEARTAG&gt;</td>
<td>Create CLEARTAG command.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>——”—”</td>
<td>&lt;Base Policy: SETPRIV&gt;</td>
<td>Create SETPRIV command.</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>——”—”</td>
<td>&lt;Base Policy: CLEARPRIV&gt;</td>
<td>Create CLEARPRIV command.</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>#</th>
<th>Instruction Type</th>
<th>Additional conditions</th>
<th>Action</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LOAD, LOAD64, STORE, STORE64, COPY, OPIMM</td>
<td>&lt;Base Policy: PROPAGATE&gt;</td>
<td>Propagate tags.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>OP</td>
<td>(but with either ARG1 or ARG2)</td>
<td>&lt;Base Policy: PROPAGATE&gt;</td>
<td>Since OP instructions have two operands, the result is tagged if either of inputs is tagged.</td>
</tr>
<tr>
<td>3</td>
<td>TAGCMD</td>
<td>&lt;Base Policy: SETTAG&gt;</td>
<td>-</td>
<td>Create SETTAG command.</td>
</tr>
<tr>
<td>4</td>
<td>—&quot;—</td>
<td>&lt;Base Policy: CLEARTAG&gt;</td>
<td>-</td>
<td>Create CLEARTAG command.</td>
</tr>
<tr>
<td>5</td>
<td>—&quot;—</td>
<td>&lt;Base Policy: GETTAG&gt;</td>
<td>-</td>
<td>Create GETTAG command.</td>
</tr>
</tbody>
</table>

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 \texttt{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.
<table>
<thead>
<tr>
<th>Test Name</th>
<th>Total # of Instructions</th>
<th>Matched # Read</th>
<th>Matched # Write</th>
<th>With Read from ARG1</th>
<th>Without Read from ARG1</th>
</tr>
</thead>
<tbody>
<tr>
<td>perlbench</td>
<td>2,687,020,357</td>
<td>72.50%</td>
<td>107.97%</td>
<td>2.32%</td>
<td>26.12%</td>
</tr>
<tr>
<td>bzip2</td>
<td>34,637,103,431</td>
<td>71.50%</td>
<td>78.84%</td>
<td>1.12%</td>
<td>28.40%</td>
</tr>
<tr>
<td>gcc</td>
<td>5,419,336,001</td>
<td>73.43%</td>
<td>111.27%</td>
<td>2.04%</td>
<td>24.99%</td>
</tr>
<tr>
<td>bwaves</td>
<td>117,419,510,623</td>
<td>59.38%</td>
<td>67.86%</td>
<td>0.39%</td>
<td>39.85%</td>
</tr>
<tr>
<td>gamess</td>
<td>242,675,586</td>
<td>53.02%</td>
<td>60.77%</td>
<td>0.91%</td>
<td>46.22%</td>
</tr>
<tr>
<td>mcf</td>
<td>3,129,160,587</td>
<td>74.80%</td>
<td>114.98%</td>
<td>1.70%</td>
<td>24.96%</td>
</tr>
<tr>
<td>milc</td>
<td>28,731,951,101</td>
<td>24.69%</td>
<td>31.42%</td>
<td>0.07%</td>
<td>74.62%</td>
</tr>
<tr>
<td>gromacs</td>
<td>35,469,684,347</td>
<td>21.69%</td>
<td>25.85%</td>
<td>0.67%</td>
<td>77.49%</td>
</tr>
<tr>
<td>cactusADM</td>
<td>13,497,373,910</td>
<td>22.00%</td>
<td>29.12%</td>
<td>0.14%</td>
<td>77.81%</td>
</tr>
<tr>
<td>namd</td>
<td>48,640,271,482</td>
<td>29.02%</td>
<td>32.49%</td>
<td>0.09%</td>
<td>70.95%</td>
</tr>
<tr>
<td>gobmk</td>
<td>69,972,013,926</td>
<td>73.68%</td>
<td>97.46%</td>
<td>1.08%</td>
<td>25.48%</td>
</tr>
<tr>
<td>dealII</td>
<td>108,862,629,102</td>
<td>59.98%</td>
<td>73.32%</td>
<td>1.23%</td>
<td>38.45%</td>
</tr>
<tr>
<td>soplex</td>
<td>95,242,541</td>
<td>59.62%</td>
<td>71.41%</td>
<td>0.80%</td>
<td>39.72%</td>
</tr>
<tr>
<td>povray</td>
<td>5,968,418,895</td>
<td>57.11%</td>
<td>78.16%</td>
<td>1.05%</td>
<td>41.35%</td>
</tr>
<tr>
<td>calculix</td>
<td>164,729,135</td>
<td>61.21%</td>
<td>74.06%</td>
<td>1.64%</td>
<td>38.03%</td>
</tr>
<tr>
<td>hmmer</td>
<td>32,572,489,558</td>
<td>61.26%</td>
<td>68.70%</td>
<td>0.91%</td>
<td>38.11%</td>
</tr>
<tr>
<td>sjeng</td>
<td>18,803,228,920</td>
<td>68.82%</td>
<td>81.77%</td>
<td>0.34%</td>
<td>38.11%</td>
</tr>
<tr>
<td>FDTD</td>
<td>7,466,145,344</td>
<td>50.11%</td>
<td>73.32%</td>
<td>0.07%</td>
<td>38.11%</td>
</tr>
<tr>
<td>libquantum</td>
<td>315,468,561</td>
<td>66.57%</td>
<td>90.01%</td>
<td>0.09%</td>
<td>33.15%</td>
</tr>
<tr>
<td>h264ref</td>
<td>106,500,405,338</td>
<td>70.95%</td>
<td>83.37%</td>
<td>0.54%</td>
<td>29.42%</td>
</tr>
<tr>
<td>tonto</td>
<td>17,696,741,780</td>
<td>50.36%</td>
<td>62.82%</td>
<td>0.95%</td>
<td>44.90%</td>
</tr>
<tr>
<td>lbm</td>
<td>2,687,020,357</td>
<td>50.11%</td>
<td>73.32%</td>
<td>1.24%</td>
<td>38.03%</td>
</tr>
<tr>
<td>omnetpp</td>
<td>1,979,754,738</td>
<td>59.62%</td>
<td>71.41%</td>
<td>0.80%</td>
<td>39.72%</td>
</tr>
<tr>
<td>astar</td>
<td>235,871,419,511</td>
<td>71.02%</td>
<td>94.30%</td>
<td>0.24%</td>
<td>28.38%</td>
</tr>
<tr>
<td>leonmp</td>
<td>35,469,684,347</td>
<td>21.69%</td>
<td>25.85%</td>
<td>0.67%</td>
<td>77.49%</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>106,500,405,338</td>
<td>21.69%</td>
<td>25.85%</td>
<td>0.67%</td>
<td>77.49%</td>
</tr>
<tr>
<td>Average</td>
<td>29,281,167,233</td>
<td>56.74%</td>
<td>73.45%</td>
<td>0.96%</td>
<td>42.36%</td>
</tr>
</tbody>
</table>

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)

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

    enum ptaxi_insn_type_t insn_type :8;
    uint8_t rs1_mask;
    uint8_t rs1_match;
    uint8_t rs1val_mask;
    uint8_t rs1val_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;

    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;
};
A.2 P-TAXI Simulator

A.2.1 Header File (ptaxisim.h)

```c
// 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);
};
```
void stop_benchmark(processor_t *p);

private:
  void print_policies(size_t context_id);
  ptaxi_insn_type_t get_insn_type(insn_t insn);
  size_t get_ptaxi_context_id(processor_t *p, bool add_if_needed);
  std::pair<ptaxi_action_t, int> determine_ptaxi_action(processor_t *p, insn_t insn, reg_t pc);
  uint8_t get_or_set_tag(processor_t *p, insn_t insn, reg_t pc, ptaxi_insn_type_t insn_type, insn_var_type_t var_type, bool set_tag, uint8_t tag_val);
  uint8_t load_tag_from_mem(processor_t *p, uint64_t addr, uint8_t rm);
  void store_tag_to_mem(processor_t *p, uint64_t addr, uint8_t rm, uint64_t val);
  tagged_reg_t v;
  // states[0] is a default policy template.
  std::vector<ptaxi_context_state_t> states;
  bool benchmark_mode = false;
  ptaxi_benchmark_counters counters;
};
#endif

Listing A.2: ptaxisim.h

A.2.2 Source File (ptaxisim.cc)

#include "ptaxisim.h"
#include "mmu.h"
#include "disasm.h"
#include "decode.h"
#include <vector>

#define ANSI_COLOR_RED  "\x1b[31m"
#define ANSI_COLOR_GREEN "\x1b[32m"
#define ANSI_COLOR_YELLOW "\x1b[33m"
#define ANSI_COLOR_BLUE  "\x1b[34m"
#define ANSI_COLOR_MAGENTA "\x1b[35m"
#define ANSI_COLOR_CYAN  "\x1b[36m"
#define ANSI_COLOR_RESET "\x1b[0m"
#define OPCODE_LOAD  (0b0000011)
#define OPCODE_LOADFP (0b0000111)
#define OPCODE_MISCMEM (0b0001111)
#define OPCODE_OPIMM (0b0010011)
#define OPCODE_AUIPC (0b0010111)
#define OPCODE_OPIMM32 (0b0011011)
#define OPCODE_STORE (0b0100011)
#define OPCODE_STOREFP (0b0100111)
#define OPCODE_AMO  (0b0101111)
# define OPCODE_OP (0b0110011)
# define OPCODE_LUI (0b0110111)
# define OPCODE_OP32 (0b0111011)
# define OPCODE_MADD (0b1000011)
# define OPCODE_MSUB (0b1000111)
# define OPCODE_NMSUB (0b1001011)
# define OPCODE_NMADD (0b1001111)
# define OPCODE_OPFP (0b1010011)
# define OPCODE_BRANCH (0b1100011)
# define OPCODE_JALR (0b1100111)
# define OPCODE_JAL (0b1101111)
# define OPCODE_SYSTEM (0b1110011)

# define OPCODE_TAGCMD (0b0001011)
# define OPCODE_TAGPOLICY (0b0101011)

# define REG_SP 2

# define TAG_RET_FROM_JAL 1
# define TAG_RET_FROM_MEM 2
# define SR_TAG_SHIFT 9
# define PTAXI_DEBUG_MODE_CONTEXT_ID 42 // Any number > 0 is fine here, just for debugging purpose.

// Adapted from https://stackoverflow.com/questions/1941307/c-debug-print-macros
#define PTAXI_VERBOSE
// # define PTAXI_DEBUG
ifdef PTAXI_VERBOSE
#define DPRINTF(fmt, args...) printf(fmt, ## args)
#else
#define DPRINTF(fmt, args...)
#endif

ifdef PTAXI_DEBUG
#define DDPRINTF(fmt, args...) printf(fmt, ## args)
#else
#define DDPRINTF(fmt, args...)
#endif

ptaxi_sim_t::ptaxi_sim_t() {
    struct ptaxi_context_state_t default_state;
    default_state.is_enable = false;
    default_state.priv_bits = 0;
    default_state.lowest_sp_addr = 0;
    states.push_back(default_state);
}

ptaxi_insn_type_t ptaxi_sim_t::get_insn_type(insn_t insn) {
    switch (insn.opcode()) {
    case OPCODE_LOAD:
        if (insn.rm() == 3) {
            return PTAXI_INSN_TYPE_LOAD64;
        }
    }
case OPCODE_STORE:
    if (insn.rm() == 3) {
        return PTAXI_INSN_TYPE_STORE64;
    } else {
        return PTAXI_INSN_TYPE_STORE;
    }

case OPCODE_OP:
    return PTAXI_INSN_TYPE_OP;

case OPCODE_OPIMM:
    if (insn.rm() == 0 && insn.i_imm() == 0) {
        return PTAXI_INSN_TYPE_COPY;
    }
    return PTAXI_INSN_TYPE_OPIMM;

case OPCODE_JAL:
    return PTAXI_INSN_TYPE_JAL;

case OPCODE_JALR:
    // rs1 == X_RA (X_RA = 1)
    if (insn.i_imm() == 0 && insn.rs1() == 1 && insn.rm() == 0 &&
        insn.rd() == 0) {
        return PTAXI_INSN_TYPE_RETURN;
    }
    return PTAXI_INSN_TYPE_JALR;

case OPCODE_TAGCMD:
    return PTAXI_INSN_TYPE_TAGCMD;

case OPCODE_TAGPOLICY:
    return PTAXI_INSN_TYPE_TAGPOLICY;

default:
    return PTAXI_INSN_TYPE_UNKNOWN;
}

size_t ptaxi_sim_t::get_ptaxi_context_id(processor_t *p, bool add_if_needed) {
    size_t context_id;
    if (benchmark_mode) {
        context_id = PTAXI_DEBUG_MODE_CONTEXT_ID;
    } else {
        context_id = (p->get_pcr(CSR_STATUS) & SR_TAG) >> SR_TAG_SHIFT;
    }
    if (add_if_needed && context_id == 0) {
        reg_t old = p->get_pcr(CSR_STATUS);
        context_id = states.size();
        if (context_id >= (1 << 7)) {
            DPRINTF("Context ID Full...\n");
            return 0;
        }
        p->set_pcr(CSR_STATUS, old | (context_id << SR_TAG_SHIFT));
    }
    while (context_id >= states.size()) {
        }
states.push_back(states[0]);
}
return context_id;

void print_insn(processor_t *p, const char *str, insn_t insn) {
  disassembler_t *disas = p->get_disassembler();
  printf("%1b[32m%s: % -25s", str, disas->disassemble(insn).c_str());
  printf("RS1: %2lu, RS2: %2lu, IMM: %8ld, RS1VAL: %8lu (0x%8lx),
          RS2VAL: %8lu (0x%8lx)\x1b[0m\n",
         insn.rs1(), insn.rs2(), insn.i_imm(), RS1, RS1, RS2, RS2);
}

std::pair<ptaxi_action_t, int> ptaxi_sim_t::determine_ptaxi_action(
    processor_t *p, insn_t insn, reg_t pc) {
  size_t context_id = get_ptaxi_context_id(p, false);
  if (context_id == 0 || !states[context_id].is_enable || IS_SUPERVISOR) {
    return std::make_pair(0, -2);
  }

  ptaxi_insn_type_t insn_type = get_insn_type(insn);
  ptaxi_action_t action = 0;
  uint8_t tag_arg1 = 0, tag_arg2 = 0, tag_out = 0, tag_out_updated = 0;
  bool is_load_tag_arg1 = false, is_load_tag_arg2 = false,
          is_load_tag_out = false;
  bool has_match = false;
  size_t i;

  for (i = 0; i < states[context_id].policy_contexts.size(); i++) {
    struct ptaxi_policy_t policy = states[context_id].
       policy_contexts[i].policy;
    bool match = (insn_type == policy.insn_type);
    if (match & policy.rs1_mask) {
      match = match & ((insn.rs1() & policy.rs1_mask) == policy.
                      rs1_match);
    }
    if (match & policy.rs2_mask) {
      match = match & ((insn.rs2() & policy.rs2_mask) == policy.
                      rs2_match);
    }
    if (match & policy.priv_mask) {
      match = match & ((states[context_id].priv_bits & policy.
                      priv_mask) == policy.priv_match);
    }
    if (match & policy.rs1val_mask) {
      match = match & ((RS1 & policy.rs1val_mask) == policy.
                      rs1val_match);
    }
  }

if (match && policy.rs2val_mask) {
    match = match && ((RS2 & policy.rs2val_mask) == policy.rs2val_match);
}

if (match && policy.tag_arg1_mask) {
    if (!is_load_tag_arg1) {
        is_load_tag_arg1 = true;
        tag_arg1 = get_or_set_tag(p, insn, pc, insn_type, INSN_ARG1, false, 0);
        if (benchmark_mode) {
            counters.tag_read++;
        }
    }
    match = match && ((tag_arg1 & policy.tag_arg1_mask) == policy.tag_arg1_match);
}

if (match && policy.tag_arg2_mask) {
    if (!is_load_tag_arg2) {
        is_load_tag_arg2 = true;
        tag_arg2 = get_or_set_tag(p, insn, pc, insn_type, INSN_ARG2, false, 0);
        if (benchmark_mode) {
            counters.tag_read++;
        }
    }
    match = match && ((tag_arg2 & policy.tag_arg2_mask) == policy.tag_arg2_match);
}

if (match && (policy.tag_out_mask || policy.tag_out_tomodify)) {
    if (!is_load_tag_out) {
        is_load_tag_out = true;
        tag_out = get_or_set_tag(p, insn, pc, insn_type, INSN_OUT, false, 0);
        tag_out_updated = tag_out;
        if (benchmark_mode) {
            counters.tag_read++;
        }
    }
    match = match && ((tag_out & policy.tag_out_mask) == policy.tag_out_match);
}

if (match) {
    has_match = true;
    struct ptaxi_policy_context_t & policy_context = states[context_id].policy_contexts[i];
    policy_context.match_count ++;
    if (policy_context.match_count <= policy_context.policy.ignore_count) {
        continue;
    }
}
{ 
tag_out_updated = ((tag_out_updated & (~policy.
tag_out_tomodify)) | policy.tag_out_set);
    if (policy.priv_tomodify) {
        states[context_id].priv_bits = ((states[context_id].
        priv_bits & (~policy.priv_tomodify))
        | policy.priv_set);
        DPRINTF("Priv Bits set to %d\n", (int) states[context_id].
        priv_bits);
    }

    action |= policy.action;
    if (policy.action == PTAXI_ACTION_BLOCK || policy.action ==
    PTAXI_ACTION_ALLOW) {
        break;
    }
}

bool real_tag_update = false;
if (is_load_tag_out && (tag_out != tag_out_updated)) {
    real_tag_update = true;
    get_or_set_tag(p, insn, pc, insn_type, INSN_OUT, true,
    tag_out_updated);
}

if (benchmark_mode) {
    uint8_t bits = (((uint8_t) is_load_tag_arg1) << 3) + (((uint8_t) is_load_tag_arg2) << 2) +
    (((uint8_t) is_load_tag_out) << 1) + (uint8_t)
    real_tag_update;
    if (real_tag_update) {
        counters.tag_write++;
    }
    counters.insns++;
    counters.needs[bits]++;
    if (has_match) {
        counters.match_insns++;
    }
}

return std::make_pair(action, i);
}

reg_t ptaxi_sim_t::execute_insn(processor_t *p, reg_t pc,
    insn_fetch_t fetch) {
    insn_t insn = fetch.insn;
    uint64_t before_tag_val = 0;
    ptaxi_insn_type_t insn_type = get_insn_type(insn);
    if (insn_type == PTAXI_INSN_TYPE_TAGCMD && insn.rd() != 0) {
        before_tag_val = TAG_S2;
    }
std::pair<ptaxi_action_t, int> paction = determine_ptaxi_action(p, insn, pc);
ptaxi_action_t action = paction.first;
disassembler_t* disas = p->get_disassembler();

if (!benchmark_mode) {
    if (action & PTAXI_ACTION_DEBUG_LINE) {
        printf(ANSI_COLOR_CYAN "%p: %-25s DEBUG\n" ANSI_COLOR_RESET, (void*) pc,
            disas->disassemble(insn).c_str());
    }

    if (action & PTAXI_ACTION_DEBUG_DETAIL) {
        size_t context_id = get_ptaxi_context_id(p, true);
        printf(ANSI_COLOR_MAGENTA "PTAXI_ACTION_DEBUG_DETAIL: %s\n",
            disas->disassemble(insn).c_str());
        printf("PC: %lx, Exit Rule: %d, Context ID: %lu\n", pc,
            paction.second, context_id);
        print_insn(p, “INSN”, fetch.insn);
        print_policies(context_id);
        printf(ANSI_COLOR_RESET);
    }

    if (action & PTAXI_ACTION_BLOCK) {
        size_t context_id = get_ptaxi_context_id(p, true);

        printf(ANSI_COLOR_MAGENTA "PTAXI_ACTION_BLOCK: %s\n"
            ANSI_COLOR_RESET,
            disas->disassemble(insn).c_str());
        print_policies(context_id);
        throw trap_tag_violation();
        return pc;
    }

    if (action & PTAXI_ACTION_GC) {
        size_t context_id = get_ptaxi_context_id(p, false);
        uint64_t cur_sp = STATE.XPR[REG_SP];
        uint64_t lowest = states[context_id].lowest_sp_addr;
        DPRINTF(ANSI_COLOR_MAGENTA "%10p: %-25s GCSTAR (-) = %p %p\n"
            ANSI_COLOR_RESET, (void*) pc,
            disas->disassemble(insn).c_str(), (void*) cur_sp, (void*) lowest);
        uint64_t clean_from = lowest - 8;
        uint64_t clean_to = cur_sp - 8;
        uint64_t clean_at;
        for(clean_at = clean_from; clean_at < clean_to; clean_at += 8) {
            MMU.store_tagged_uint64(clean_at, 0, 0);
        }
        DPRINTF(ANSI_COLOR_GREEN "CLEAN FROM %p to %p\n"
            ANSI_COLOR_RESET, (void*) clean_from,
(void *) clean_to);

    states[context_id].lowest_sp_addr = cur_sp;
}

if (insn_type == PTAXI_INSN_TYPE_TAGCMD && insn.rd() != 0) {
    if (action & PTAXI_ACTION_GETTAG) {
        DDPRINTF(ANSI_COLOR_CYAN "%10 p: % -25s GETTAG (%2 d) = %d\n"
            ANSI_COLOR_RESET, (void *) pc, 
            disas->disassemble(insn).c_str(), (int) insn.rs2(), (int)
            before_tag_val);
        WRITE_RD(before_tag_val);
    } else {
        WRITE_RD(RS2);
    }
}

reg_t res = fetch.func(p, insn, pc);

if(insn.rd() == REG_SP && !IS_SUPERVISOR) {
    size_t context_id = get_ptaxi_context_id(p, false);
    if (context_id != 0) {
        uint64_t cur_sp = STATE.XPR[REG_SP];
        DDPRINTF(ANSI_COLOR_CYAN "%10 p: % -25s MODISP (--) = %p\n"
            ANSI_COLOR_RESET, (void *) pc, 
            disas->disassemble(insn).c_str(), (void *) cur_sp);
        if (cur_sp < states[context_id].lowest_sp_addr || states[
            context_id].lowest_sp_addr == 0) {
            states[context_id].lowest_sp_addr = cur_sp;
            DDPRINTF(ANSI_COLOR_BLUE "%10 p: % -25s LOWEST (--) = %p\n"
                ANSI_COLOR_RESET, (void *) pc, 
                disas->disassemble(insn).c_str(), (void *) cur_sp)
        };
    }
}

return res;

void ptaxi_sim_t::print_policies(size_t context_id) {
    printf("Policy Count: %lu\n-----\n", states[context_id].
policy_contexts.size());
    for (size_t i = 0; i < states[context_id].policy_contexts.size();
        i++) {
        struct ptaxi_policy_context_t policy_context = states[context_id ]
            .policy_contexts[i];
        printf("%3lu %5d %3d%3d %3lu\n", i, (int) policy_context.
policy.insn_type,
            (int) policy_context.policy.rs1val_match, (int)
policy_context.policy.action,
            policy_context.match_count);
void ptaxi_sim_t::add_policy(processor_t *p, uint64_t a, uint64_t b, uint64_t c) {
    size_t context_id = get_ptaxi_context_id(p, true);
    union ptaxi_policy_serialized ps;
    ps_regs.a = a;
    ps_regs.b = b;
    ps_regs.c = c;
    struct ptaxi_policy_context_t policy_context;
    policy_context.policy = ps.policy;
    policy_context.match_count = 0;
    states[context_id].policy_contexts.push_back(policy_context);
}

void ptaxi_sim_t::run_tag_command(processor_t *p, uint64_t cmd) {
    size_t context_id = get_ptaxi_context_id(p, true);
    if (cmd == 0) {
        DPRINTF(ANSI_COLOR_CYAN "Enforcing.. Context Id = %d
" ANSI_COLOR_RESET, (int) context_id);
        states[context_id].is_enable = true;
    } else {
        DPRINTF(ANSI_COLOR_YELLOW "TAG COMMAND %lu
" ANSI_COLOR_RESET, cmd);
    }
    #ifdef PTAXI_VERBOSE
    print_policies(context_id);
    #endif
}

uint8_t ptaxi_sim_t::get_or_set_tag(processor_t *p, insn_t insn, reg_t pc, ptaxi_insn_type_t insn_type, insn_var_type_t var_type, bool set_tag, uint8_t tag_val) {
    bool is_invalid = false, is_mem = false;
    uint64_t addr;
    switch (insn_type) {
    case PTAXI_INSN_TYPE_LOAD64: // arg1 = MEM, arg2 = N/A, out = REG
    case PTAXI_INSN_TYPE_LOAD:
        if (var_type == INSN_ARG1) {
            is_mem = true;
            addr = RS1 + insn.i_imm();
        } else if (var_type == INSN_ARG2) {
            is_invalid = true;
        } else {
            reg = insn.rd();
        }
        break;
case PTAXI_INSN_TYPE_STORE64: // arg1 = REG, arg2 = N/A, out = MEM  
   case PTAXI_INSN_TYPE_STORE:  
      if (var_type == INSN_ARG1) {  
         reg = insn.rs2();  
      } else if (var_type == INSN_ARG2) {  
         is_invalid = true;  
      } else {  
         is_mem = true;  
         addr = RS1 + insn.s_imm();  
      }  
   break;  
   case PTAXI_INSN_TYPE_TAGCMD:  
   case PTAXI_INSN_TYPE_OP: // arg1 = REG1, arg2 = REG2, out = REGOUT  
      if (var_type == INSN_ARG1) {  
         reg = insn.rs1();  
      } else if (var_type == INSN_ARG2) {  
         reg = insn.rs2();  
      } else {  
         reg = insn.rd();  
      }  
   break;  
   case PTAXI_INSN_TYPE_OPIMM: // arg1 = REG1, arg2 = N/A, out = REGOUT  
   case PTAXI_INSN_TYPE_COPY:  
      if (var_type == INSN_ARG1) {  
         reg = insn.rs1();  
      } else if (var_type == INSN_ARG2) {  
         is_invalid = true;  
      } else {  
         reg = insn.rd();  
      }  
   break;  
   case PTAXI_INSN_TYPE_JAL: // arg1 = TARGET, arg2 = n/a, arg3 = REGOUT  
      if (var_type == INSN_ARG1) {  
         /*is_mem = true;  
         addr = pc + insn.uj_imm();*/  
         is_invalid = true;  
      } else if (var_type == INSN_ARG2) {  
         is_invalid = true;  
      } else {  
         reg = insn.rd();  
      }  
   break;  
   case PTAXI_INSN_TYPE_JALR: // arg1 = REG1, arg2 = TARGET, arg3 = REGOUT  
   case PTAXI_INSN_TYPE_RETURN:  
      if (var_type == INSN_ARG1) {  
         reg = insn.rs1();  
      } else if (var_type == INSN_ARG2) {  
         is_mem = true;
addr = (RS1 + insn.i_imm()) & ~reg_t(1);
}
else {
    reg = insn.rd();
}
break;
default:
is_invalid = true;
break;

if (is_invalid) {
    DPRINTF("get_or_set_tag: ISINVALID TRAP!\n");
    DPRINTF("GET OR SET TAG %lx %d %d %d\n", insn.bits(), (int) insn_type, (int) var_type,
    (int) set_tag);
    throw trap_tag_violation();
    return 0;
}

disassembler_t* disas = p->get_disassembler();

if (is_mem) {
    if (set_tag) {
        store_tag_to_mem(p, addr, insn.rm(), tag_val);
        DPRINTF(ANSI_COLOR_CYAN "\%10p: %25s SETMEM (%p) = %d\n"
        ANSI_COLOR_RESET, (void*) pc, disas->disassemble(insn).c_str(), (void*) addr, (int) tag_val);
    } else {
        uint8_t tag_val_from_mem = load_tag_from_mem(p, addr, insn.rm());
        DPRINTF(ANSI_COLOR_CYAN "\%10p: %25s LOADTG (%p) = %d\n"
        ANSI_COLOR_RESET, (void*) pc, disas->disassemble(insn).c_str(), addr, (int) tag_val_from_mem);
        return tag_val_from_mem;
    }
} else {
    if (reg == 0) {
        return 0;
    }
    if (set_tag) {
        DPRINTF(ANSI_COLOR_CYAN "\%10p: %25s SETREG (%2d) = %d\n"
        ANSI_COLOR_RESET, (void*) pc, disas->disassemble(insn).c_str(), (int) reg, (int) tag_val);
        STATE.XPR.write_tag(reg, tag_val);
    } else {
        return STATE.XPR.read_tag(reg);
    }
}
return 0;
uint8_t ptaxi_sim_t::load_tag_from_mem(processor_t *p, uint64_t addr, uint8_t rm) {
    switch (rm) {
    case 0: // LB
        return MMU.load_tag_only_int8(addr);
    case 1: // LH
        return MMU.load_tag_only_int16(addr);
    case 2: // LW
        return MMU.load_tag_only_int32(addr);
    case 3: // LD
        return MMU.load_tag_only_int64(addr);
    case 4: // LBU
        return MMU.load_tag_only_uint8(addr);
    case 5: // LHU
        return MMU.load_tag_only_uint16(addr);
    case 6: // LWU
        return MMU.load_tag_only_uint32(addr);
    default:
        DPRINTF("get_or_set_tag: ISINVALID TRAP2!");
        throw trap_tag_violation();
        return 0;
    }
}

void ptaxi_sim_t::store_tag_to_mem(processor_t *p, uint64_t addr, uint8_t rm, uint64_t val) {
    switch (rm) {
    case 0: // SB
        MMU.store_tag_only_uint8(addr, val);
        break;
    case 1: // SH
        MMU.store_tag_only_uint16(addr, val);
        break;
    case 2: // SW
        MMU.store_tag_only_uint32(addr, val);
        break;
    case 3: // SD
        MMU.store_tag_only_uint64(addr, val);
        break;
    default:
        DPRINTF("get_or_set_tag: ISINVALID TRAP3!");
        throw trap_tag_violation();
        break;
    }
}

void ptaxi_sim_t::start_benchmark(processor_t *p) {
    if (benchmark_mode) {
        return;
    }
    DPRINTF(ANSI_COLOR_GREEN "Start Benchmark...
" ANSI_COLOR_RESET);
    memset(&counters, 0, sizeof(counters));
benchmark_mode = true;

void ptaxi_sim_t::stop_benchmark(processor_t *p) {
    if (!benchmark_mode) {
        return;
    }
    DPRINTF(ANSI_COLOR_GREEN "Stop Benchmark..\n" ANSI_COLOR_RESET);
    print_policies(PTAXI_DEBUG_MODE_CONTEXT_ID);
    printf("RESULT,%lu,%lu,%lu,%lu", counters.insns, counters.
    match_insns, counters.tag_read, counters.tag_write);
    for(int i = 0; i < 16; i++) {
        printf(",%lu", counters.needs[i]);
    }
    printf("\n");
    benchmark_mode = false;
}

Listing A.3: ptaxisim.cc

A.3 User-level Libraries

A.3.1 Basic Interface (ptaxi_user.h)

ifndef _PTAXI_USER_H
#define _PTAXI_USER_H

#include <stdint.h>
#include <string.h>

#include "ptaxi_common.h"

// Use macro so that we don’t have to set previous return pointers.
void ptaxi_tag_command(code) {
    __asm__ ("tagcmd zero,%0,zero" :: "r"(code));
}

void ptaxi_enforce_policy() {
    ptaxi_tag_command(0);
}

void ptaxi_add_raw_policy(uint64_t a, uint64_t b, uint64_t c) {
    __asm__ volatile ("tagpolicy %2,%0,%1" :: "r"(a), "r"(b), "r"(c));
}

void ptaxi_add_policy(struct ptaxi_policy_t policy) {
    union ptaxi_policy_serialized ps;
    ps.policy = policy;
    ptaxi_add_raw_policy(ps.regs.a, ps.regs.b, ps.regs.c);
}
A.3.2 Header File for Inclusion by User-Level Applications (ptaxi.h)

Listing A.4: ptaxi_user.h

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

```c
#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_propagate_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;
```
memset(&default_policy, 0, sizeof(default_policy));
default_policy.tag_out_tomodify = tagbit;
default_policy.insn_type = insn_type;

if (arg1) {
    policy = default_policy;
    policy.tag_arg1_mask = tagbit;
    policy.tag_arg1_match = tagbit;
    policy.tag_out_set = tagbit;
    ptaxi_add_policy(policy);
}

if (arg2) {
    policy = default_policy;
    policy.tag_arg2_mask = tagbit;
    policy.tag_arg2_match = tagbit;
    policy.tag_out_set = tagbit;
    ptaxi_add_policy(policy);
}

policy = default_policy;
policy.tag_out_set = 0;
if (arg1) {
    policy.tag_arg1_mask = tagbit;
    policy.tag_arg1_match = 0;
}
if (arg2) {
    policy.tag_arg2_mask = tagbit;
    policy.tag_arg2_match = 0;
}
ptaxi_add_policy(policy);

void ptaxi_base_policy_clear_by_type(uint8_t tagbit, enum ptaxi_insn_type_t insn_type) {
    struct ptaxi_policy_t policy;
    memset(&policy, 0, sizeof(policy));
    policy.tag_out_tomodify = tagbit;
    policy.insn_type = insn_type;
    policy.tag_out_set = 0;
    ptaxi_add_policy(policy);
}

void ptaxi_base_policy_create_settag(uint8_t tagbit) {
    struct ptaxi_policy_t policy;
    memset(&policy, 0, sizeof(policy));
    policy.insn_type = PTAXI_INSN_TYPE_TAGCMD;
    policy.rsival_mask = 0b11111111;
    policy.rsival_match = PTAXI_TAGCMD_PREFIX_SETTAG + log2bit(tagbit);
    policy.tag_out_tomodify = tagbit;
policy.tag_out_set = tagbit;
ptaxi_add_policy(policy);
}

void ptaxi_base_policy_create_cleartag(uint8_t tagbit) {
    struct ptaxi_policy_t policy;
    memset(&policy, 0, sizeof(policy));
    policy.tag_out_tomodify = tagbit;
    policy.insn_type = PTAXI_INSN_TYPE_TAGCMD;
    policy.rs1val_mask = 0b11111111;
    policy.rs1val_match = PTAXI_TAGCMD_PREFIX_CLEARTAG + log2bit(tagbit);
    policy.tag_out_set = 0;
    ptaxi_add_policy(policy);
}

void ptaxi_base_policy_create_gettag(uint8_t tagbit) {
    struct ptaxi_policy_t policy;
    memset(&policy, 0, sizeof(policy));
    policy.tag_out_tomodify = tagbit;
    policy.insn_type = PTAXI_INSN_TYPE_TAGCMD;
    policy.rs1val_mask = 0b11111111;
    policy.rs1val_match = PTAXI_TAGCMD_PREFIX_GETTAG + log2bit(tagbit);
    policy.tag_out_set = 0;
    policy.action = PTAXI_ACTION_GETTAG;
    ptaxi_add_policy(policy);
}

void ptaxi_base_policy_create_setpriv(uint8_t tagbit) {
    struct ptaxi_policy_t policy;
    memset(&policy, 0, sizeof(policy));
    policy.insn_type = PTAXI_INSN_TYPE_TAGCMD;
    policy.rs1val_mask = 0b11111111;
    policy.rs1val_match = PTAXI_TAGCMD_PREFIX_SETPRIV + log2bit(tagbit);
    policy.priv_tomodify = tagbit;
    policy.priv_set = tagbit;
    ptaxi_add_policy(policy);
}

void ptaxi_base_policy_create_clearpriv(uint8_t tagbit) {
    struct ptaxi_policy_t policy;
    memset(&policy, 0, sizeof(policy));
    policy.insn_type = PTAXI_INSN_TYPE_TAGCMD;
    policy.rs1val_mask = 0b11111111;
    policy.rs1val_match = PTAXI_TAGCMD_PREFIX_CLEARPRIV + log2bit(tagbit);
    policy.priv_tomodify = tagbit;
    policy.priv_set = 0;
    ptaxi_add_policy(policy);
}
uint8_t ptaxi_base_policy_gettag(uint8_t tagbit, void *addr) {
    uint64_t out;
    uint64_t cmd = PTAXI_TAGCMD_PREFIX_GETTAG + log2bit(tagbit);
    uint64_t in = *((uint64_t *) addr);
    __asm__ volatile("tagcmd %0,%2,%1" : "=r"(out) : "r"(in), "r"(cmd) );
    return out;
}

// tagval is a boolean. True = set/False = clear
void ptaxi_base_policy_settag(uint8_t tagbit, void *addr, uint8_t tagval) {
    uint64_t cmd;
    if (tagval > 0) {
        cmd = PTAXI_TAGCMD_PREFIX_SETTAG + log2bit(tagbit);
    } else {
        cmd = PTAXI_TAGCMD_PREFIX_CLEARTAG + log2bit(tagbit);
    }
    uint64_t *raddr = (uint64_t *) addr;
    uint64_t in = *raddr;
    uint64_t out = 0;
    __asm__ volatile("tagcmd %0,%2,%1" : "=r"(out) : "r"(in), "r"(cmd) );
    *raddr = out;
}

void ptaxi_base_policy_settag_multi(uint8_t tagbit, void *addr, size_t size, uint8_t tagval) {
    size_t block = size/4;
    uint64_t * pos = (uint64_t *) addr;
    size_t i;
    for (i = 0; i < block; i++) {
        ptaxi_base_policy_settag(tagbit, addr, tagval);
        pos++;
    }
}

// privval is a boolean. True = set/False = clear
void ptaxi_base_policy_setpriv(uint8_t tagbit, uint8_t privval) {
    uint8_t cmd;
    if (privval > 0) {
        cmd = PTAXI_TAGCMD_PREFIX_SETPRIV + log2bit(tagbit);
    } else {
        cmd = PTAXI_TAGCMD_PREFIX_CLEARPRIV + log2bit(tagbit);
    }
    ptaxi_tag_command(cmd);
}
#endif

Listing B.1: ptaxi_base_policy.h
B.1.2 Return Address Protection

```c
#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_propagate_by_type(tagbit, PTAXI_INSN_TYPE_STORE64, 1, 0);
    ptaxi_base_policy_propagate_by_type(tagbit, PTAXI_INSN_TYPE_LOAD64, 1, 0);
    ptaxi_base_policy_propagate_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)

```c
#ifndef _PTAXI_POLICY_PRIVILEGE_H
#define _PTAXI_POLICY_PRIVILEGE_H
```

97
void ptaxi_policy_privilege_init(uint8_t tagbit) {
    ptaxi_base_policy_create_settag(tagbit);
    ptaxi_base_policy_create_gettag(tagbit);
    ptaxi_base_policy_create_cleartag(tagbit);
    ptaxi_base_policy_create_setpriv(tagbit);
    ptaxi_base_policy_create_clearpriv(tagbit);
    ptaxi_base_policy_propagate_by_type(tagbit, PTAXI_INSN_TYPE_STORE64, 1, 0);
    ptaxi_base_policy_propagate_by_type(tagbit, PTAXI_INSN_TYPE_LOAD64, 1, 0);
    ptaxi_base_policy_propagate_by_type(tagbit, PTAXI_INSN_TYPE_COPY, 1, 0);
}

void ptaxi_policy_privilege_protect_tag_from_nonprivilege(uint8_t tagbit) {
}

void ptaxi_policy_privilege_protect_data(uint8_t tagbit) {
    struct ptaxi_policy_t default_policy, policy;
    memset(&default_policy, 0, sizeof(default_policy));
    default_policy.priv_mask = tagbit;
    default_policy.priv_match = 0;
    default_policy.action = PTAXI_ACTION_BLOCK;

    policy = default_policy;
    policy.insn_type = PTAXI_INSN_TYPE_LOAD;
    policy.tag_arg1_mask = tagbit;
    policy.tag_arg1_match = tagbit;
    ptaxi_add_policy(policy);

    policy = default_policy;
    policy.insn_type = PTAXI_INSN_TYPE_LOAD64;
    policy.tag_arg1_mask = tagbit;
    policy.tag_arg1_match = tagbit;
    ptaxi_add_policy(policy);

    policy = default_policy;
    policy.insn_type = PTAXI_INSN_TYPE_STORE;
    policy.tag_out_mask = tagbit;
    policy.tag_out_match = tagbit;
    ptaxi_add_policy(policy);

    policy = default_policy;
    policy.insn_type = PTAXI_INSN_TYPE_STORE64;
    policy.tag_out_mask = tagbit;
    policy.tag_out_match = tagbit;
    ptaxi_add_policy(policy);
}

void ptaxi_policy_privilege_enter(uint8_t tagbit) {
    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

Listing B.4: ptaxi_policy_gc.h

B.1.5 Call Debugging

Listing B.5: ptaxi_policy_debug_call.h
B.2 Test Cases

B.2.1 Return Address Protection

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

define TAG_RETURNADDRESS 1

goattribute ((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;
}
```
B.2.2 Get and Set Tags

```c
#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, 0);
    printf("A = %lu, TAG(A) = %d (should be 0)\n", a, (int)
        ptaxi_base_policy_gettag(TAGBIT, &a));
    return 0;
}
```

B.2.3 Malloc with Memory Compartmentalization

```c
#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;
```
malloc_ll *malloc_root;

void __attribute__ (( constructor )) ptaxi_app_policy () {
    ptaxi_policy_privilege_protect_data (MALLOC_TAGBIT);
    ptaxi_policy_privilege_init (MALLOC_TAGBIT);
    ptaxi_enforce_policy ();
}

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;
}

void read_size (char *a) {
    malloc_ll *m = (malloc_ll *) (a - sizeof (malloc_ll));
    size_t size = m->size;  // Should trap here!
    printf ("Malloc Size = %d \n", size);
}

int main (int argc, char ** argv) {
    char *a = (char *) pmalloc (48);
    a[0] = 'H';
    a[1] = 'E';
    a[2] = 'L';
    a[3] = 'L';
    a[4] = 'O';
    a[5] = '0';
    printf ("String = %s\n", a);
    if (argc > 1) {
        // Should pass instead of fail.
    } else {
        read_size (a);
    }
}
B.2.4 Basic Taint Tracking

Listing B.8: test_simple_malloc.c

```c
#include <stdio.h>
#include <unistd.h>
#include "ptaxi.h"
#define TAINT_TAGBIT 1

void __attribute__ (( constructor )) ptaxi_app_policy () {
    ptaxi_policy_privilege_init ( TAINT_TAGBIT );
    ptaxi_base_policy_propagate_by_type ( TAINT_TAGBIT,
            PTAXI_INSN_TYPE_OP , 1, 1);
    ptaxi_base_policy_propagate_by_type ( TAINT_TAGBIT,
            PTAXI_INSN_TYPE_OPIMM, 1, 0);
    ptaxi_enforce_policy();
}

uint64_t get_unfiltered_input () {
    uint64_t input = 42;
    ptaxi_base_policy_settag ( TAINT_TAGBIT , (void *) (&input), 1);
    printf ( " DEBUGE : %d\n", (int) ptaxi_base_policy_gettag ( TAINT_TAGBIT,
        (void *) (&input)));
    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 B.9: test_taint_tracking.c

B.2.5 Stack Garbage Collection

Listing B.8: test_simple_malloc.c

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

Listing B.9: test_taint_tracking.c

B.2.5 Stack Garbage Collection

Listing B.8: test_simple_malloc.c

```c
#include <stdio.h>
#include "ptaxi.h"
```
uint64_t *gsecretnumberptr = NULL;

int i;

int g(int x) {
    return x * 2 + 9;
}

int f() {
    ptaxi_tag_command(123);
    uint64_t secretnumber = 0xDEADBEEF;
    gsecretnumberptr = &secretnumber;
    int i, s = 0;
    for (i = 0; i < 20; i++) {
        s += g(i) + secretnumber;
    }
    ptaxi_tag_command(456);
    return s;
}

int main(int argc, char** argv) {
    if (!(argc > 1)) {
        ptaxi_policy_gc();
        ptaxi_enforce_policy();
    } else {
        printf("Should show deadbeef\n");
    }
    ptaxi_tag_command(7);
    f();
    ptaxi_tag_command(8);
    printf("Secret = %x at %p\n", *gsecretnumberptr, (void *)gsecretnumberptr);
    return 0;
}

Listing B.10: test_gc.c
Bibliography


106


