2015-06-26

Kernel-assisted Pattern Analysis of Memory Events

Laing, Sarah

UNIVERSITY OF CALGARY

Kernel-assisted Pattern Analysis of Memory Events

by

Sarah Laing

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

June, 2015

Abstract

Memory interception is used to create a record of a program's execution. Filtering the intercepted memory events enables one to find patterns in the memory accesses of a target program, patterns that can be used to find errors or vulnerabilities in the program.

We present Cage, a kernel-level mechanism for intercepting and filtering the memory events of a user-level process. Cage uses a technique that generates a page fault for every instruction level memory access. The filtering component of Cage extends and uses the Berkeley Packet Filter infrastructure to filter memory events that have been intercepted. In the page fault handler, information related to the memory event is composed into a packet-like format and exported over a specialized memory network device. Standard network packet capture tools such as Wireshark can be used to capture from the memory network device to retrieve the information about each memory event.

Table of Contents

List of Figures and Illustrations

Chapter 1

Introduction

Modern computers use a form of volatile or non-persistent storage - memory - to store information that is necessary for the computer to run but that does not need to be saved. Computers use memory for just about everything. Every program that runs uses its own section of memory to store its data and instructions that tell the program what to do. Every operation a program performs can contain multiple instructions. Each instruction must be read from memory before it can be executed. Instructions can also read or write to a program's data generating yet more memory accesses.

Figure 1.1 is a simple C program that just prints the value of a piece of program data. The program data labelled as "variable" on line one is stored in the memory of the program. Both printing the contents of this program's data, and returning its value on lines three and four respectively, requires that memory be accessed and read. Furthermore, the "main" function declaration on line two writes more data in the memory of the program and reads more data from memory. The "printf" statement on line three is a call to a shared library. This shared library has its own section of memory that must be read from and written to. Each line of this simple program is translated into multiple low-level instructions. This requires that multiple instructions must be read from memory in order to execute each line of this program.

```
1.   int variable = 0;
2.   int main(int argc, char *argv[]){
3.      printf("Variable = %d\n", variable);
4.      return variable;
5.   }
```

Figure 1.1: An example C program.

In addition, there is start up and tear down code that is executed for every program that is not shown here. This additional code must also read and write from memory. Our simple test program produces 108120 memory events in its entirety where a "memory event" is a read or write operation to memory. This shows the magnitude of memory events that are generated, even by seemingly simple programs.

1.1 Motivation

Memory interception is a technique that monitors a target program and watches all of the memory events that the target program generates. The motivation behind memory interception is that it provides a method of determining exactly what a program is doing during its execution. The combination of instruction fetch memory events and data accesses shows a clear picture of the control flow path the program is taking as well as the contents of the data that brought the program along this control flow path. The memory events of a process form a record of its execution. This record can be used for many purposes from debugging to vulnerability detection.

The memory event stream is the combination of all of the memory events generated by a program. As demonstrated by our simple program in Figure 1.1, the memory event stream can contain a large number of memory events even for a small program. To deal with the volume of memory events, we filter the memory event stream. The motivation behind filtering the memory event stream is that it provides a method of detecting memory errors or vulnerabilities. Different errors or vulnerabilities present as different patterns within the memory event stream. For example, a double fetch of a piece of data within the same semantic context can be indicative of a "time of check to time of use" vulnerability where the data may have been modified between the two fetches. This presents as a specific pattern in the memory event stream that a filter can detect.

1.2 Overview of Problem

Memory interception exists as a challenging problem in computer science. The ability to intercept the read and write memory events of a process has been attempted many times. Many solutions to intercepting the memory events of a process exist such as creating specialized hardware [31, 54, 53], emulation [4, 45, 18], or instrumentation [6, 33]. Each of these solutions has some drawbacks associated with it such as reliability or performance. The memory events generated by a process is essentially a high rate, high event data stream. It is the speed with which the memory events occur and the magnitude of these memory events that makes intercepting and capturing these memory events difficult.

Intercepting and capturing the memory events of a process is only half the problem. The other half is dealing with the large volume and velocity of data that is produced. A process produces a memory event each time it fetches an instruction to be executed and each time it reads or writes to a location in memory. These are memory events produced at the instruction level. Intercepting memory events at the microcode level or from the cache is outside the scope of this thesis. Larger, more complex programs would logically produce more memory events than simpler programs. Bzip2 for example, produces over 3 billion memory events from the stack alone. This large amount of data must be filtered in some fashion in order to pick out the memory events of interest. Many of the filtering techniques rely on finding patterns within the memory event stream. These often rely on maintaining a piece of state for each portion of memory within the memory of the target process [50, 54, 33]. This has the side effect of increasing the memory consumption of the target process which is not scalable to significantly large programs.

The challenges facing memory interception and filtering of the memory event stream can be thought of as analogous to an intrusion detection system for a high

speed network. Both systems must be able to intercept all of the memory events or network packets in order to get a clear picture of what is happening. Both systems must also be able to filter the memory events or network packets and make some decision regarding the memory event or network packet. The goal of an intrusion detection system is to intercept and filter network packets at a speed that as closely resembles line speed as possible, yet still be reliable and complete. Memory interception and filtering of the memory event stream have the same goals as an intrusion detection system.

To state the problem more formally, a memory event m is a tuple of elements, $(t, a, ea, ip, dea, i, lea, lip)$ where t is the time that the memory event occurs, a is the access type of the memory event, either read or write, ea is the effective address that is being accessed by the memory event, ip is the address of the instruction being executed, dea is the data contained at the effective address, i is the instruction being executed, lea is the label for the page containing the effective address, and lip is the label for the page containing the instruction address.

A memory event stream is an ordered list of memory events such that
$M = \{m_0 \dots m_n\}$ where n is a possibly infinite number.

Memory interception is a function over M such that each memory event m_i is intercepted and captured. More information on the memory interception function can be found in Section 3.1.

A filter is a function over M that takes as input a memory event m_i from M and a state $= s$ and produces an answer $a_i = \{accept, ignore, reject\}$ and a potentially new state s' for each m_i. More information on the filtering function can be found in Section 3.4.

The memory interception function and the filtering function must behave in a principled fashion. That is, they must have the following properties:

1. Speed
2. Transparency
3. Reliability
4. Completeness

The speed of both the memory interception function and the filtering function must be taken into account. In the ideal case, neither the memory interception function nor the filtering function should inhibit the runtime speed of the target process. With the exception of hardware solutions, where both the memory interception function and the filtering function are a component of the hardware, the ideal case is not possible. There will always be some amount of overhead imposed by the memory interception function and the filtering function.

The transparency of the memory interception function and the filtering function must also be taken into account. In the ideal case, neither the presence of a memory interception function nor a filtering function would modify the behaviour of the target process. This would mean that both the memory interception function and the filtering function would not be detectable by the target process through any means. Currently we know of no solution that meets this ideal case. Transparency can also mean that the memory interception function and the filtering function require no modifications to the target program's source code or binary image. This lighter form of transparency also has the goal of not modifying the behaviour of the target process but does not guarantee that the target process cannot detect the monitoring. If the memory interception function and the filtering function are not at least lightly transparent to the target process they will not be able to create the record of the target program's execution.

The reliability of the memory interception function and the filtering function means that the memory event stream is identical for multiple executions of the target

program with the same inputs and on the same hardware.[1] The memory interception function and the filtering function must reliably intercept and filter each memory event m_i from the memory event stream where i occurs at the same place in the memory event stream across executions of the target program and the tuple elements of m_i are identical. The only exception to this is the element time because it is possible for the memory event m_i to occur at a different time because of differences in the scheduling of the target process. If the memory interception function and the filtering function are not reliable they will fail at producing the record of the target program's execution.

The completeness of the memory interception function and the filtering function refer to their vantage points on the memory event stream M. In order to be considered complete the memory interception function must intercept each m_i from M. In order for the filtering function to be considered complete it must filter each m_i from M. Put more simply, the memory interception function and the filtering function must work on each memory event within the memory event stream to be considered complete. Without being considered complete, the record of the target program's execution would not be complete, which goes against the motivation for memory interception.

1.3 Overview of Cage

This thesis presents a new mechanism referred to as Cage or the Cage mechanism throughout this thesis that has two main components. The first component is a mechanism to intercept the memory events of a process and the second component is a mechanism to filter the memory events of a process. Both of these mechanisms work together to intercept the instruction level memory events of a process and to filter these memory events as they occur during the execution of a process. Both

[1] "Identical" does not take into account multi-threaded programs. We leave multi-threaded programs to future work.

mechanisms exist as a series of modifications to the 3.9.4 version of the Linux kernel as well as supporting loadable kernel modules and user-level programs.

Thesis Statement: *It is possible to create a memory interception and filtering mechanism which adheres to the aforementioned principles of being lightly transparent, reliable and complete.*

The memory interception mechanism intercepts memory events by generating a page fault for each memory event within a certain range of memory. This memory range may be as small as one page or as large as the entire process address space. This memory interception approach is lightly transparent to the target process. There are no modifications required to the target program's source code or binary image. This memory interception technique reliably and completely intercepts all of the memory events within a certain range of memory. Information about each memory event is gathered and composed into a packet-like format referred to as a memory event packet. These memory event packets can be captured and stored to form a record of the execution of the target process. There is also no additional hardware required and this memory interception technique can run on commodity hardware and operating systems.

The Cage filtering mechanism extends and uses a well-known filtering tool called the Berkeley Packet Filter. This filtering mechanism does not require that state be stored within the memory of a target process. The filtering occurs during the page faults that the memory interception mechanism generates. This allows the ability to filter memory events as they occur. The location of the filtering mechanism allows a decision to be made about a memory event as it occurs. This grants the opportunity to disallow a memory event that violates the filter and terminate the target process accordingly. The location of the filtering mechanism also means that it remains lightly transparent to the target process and requires no modifications to the target

program's source code. The location of the filtering mechanism also ensures that it is reliable and complete. As long as the memory interception mechanism is guaranteed to intercept each memory event, the filtering mechanism is guaranteed to filter each memory event.

Cage is a dual-use security tool meaning that it is not inherently offensive or defensive. Similar to other tools, such as gdb, it has many uses, some of which can be considered offensive and some of which can be considered defensive. There are multiple features of Cage. As mentioned previously, Cage can be used to intercept and filter the memory events for a range of memory as small as the size of a page to the entire process address space. Cage can target an already executing process or execute a process with a targeted range of memory, such as the stack, .text, or .data sections. Cage can also execute a process with the entire process address space targeted so that every memory event the process makes is intercepted. The filtering mechanism can filter the memory events from a single address or from an address range that can be as large as the entire process address space. Any memory event that is intercepted can be filtered. The filtering mechanism can also be disabled to allow the capturing of every memory event within the targeted area. This allows a record to be generated of exactly which memory events occurred within the targeted area.

An earlier version of this work was published at the 8^{th} USENIX Workshop on Offensive Technologies (WOOT) in 2014 [20].

1.4 Overview of Thesis

The rest of the thesis is organized as follows. Chapter 2 discusses work that is related to Cage both in terms of memory interception and memory filtering. The Cage mechanism is described in detail in Chapter 3 as well as the supporting mechanisms

that have been created to target programs. The evaluation of Cage including its performance and generality to various platforms is discussed in Chapter 4. Chapter 5 discusses the applications of Cage focusing on the filtering aspect and its capabilities and limitations. Finally, Chapter 6 concludes and discuses future work.

Chapter 2

Related Work

Two major aspects of Cage are the memory interception mechanism and the ability to find patterns in the memory event stream. Correspondingly, this chapter discusses related work in two major parts. In the first two sections, work related to memory interception techniques in general, and work most closely related to Cage's memory interception mechanism is discussed. The third section discusses work that uses one of the memory interception techniques and focuses on finding patterns in the memory event stream.

2.1 Memory Interception Techniques

There are numerous techniques to intercepting memory events, each with their own advantages and disadvantages. This section discusses a general overview of existing techniques beginning with standard "built-in" approaches, and continuing with hardware modifications and extracting memory events from hardware, binary instrumentation, emulation and virtualization, and the reuse of existing hardware facilities.

2.1.1 Standard or "built-in" Approaches

Most commodity platforms come with tools similar to gdb [12] and strace [38]. System call tracing via strace examines memory events at the system-call level and is a read-only view of this event stream. Library interception, using LD_PRELOAD [22], of the C library's malloc and other memory management functions offers finer granularity and the possibility of rewriting the events that strace lacks, but it still misses instruction-level memory events (this lack of fine-grained instruction-level detail is

the stated motivation behind many projects, including Fenris [52]). Basic debuggers like gdb offer fine-grained observation and control of a running program, including the memory of that program. However, they are primarily intended to provide interactive debugging sessions and therefore may require some scripting to support offline or automated analysis. The gdb debugger offers Python bindings and its own native command set [11, section 23.2 and 23.3] in support of this. Some other debuggers, such as IDA [14], offer facilities more specialized to reverse engineering, such as propagation of labels and tags to group common pieces of functionality in a program. Generally, the ptrace [32] or dtrace [41] mechanisms support the ability to implement custom debuggers or other methods of program supervision, but this requires that analysts implement their own userspace C or D code. Pin [24] and Valgrind [29] make use of the ptrace API to aid in instrumenting binaries.

2.1.2 Extracting Memory Events from Hardware and Hardware Modifications

One approach to capturing memory accesses is extracting memory events directly from the hardware. Work such as CoPILOT [31] uses an additional PCI card to request memory accesses from the DMA controller at the hardware level. Other work such as MemTracker [50] modifies the processor pipeline to intercept instruction-level load and store memory events. Accmon [53] and iWatcher [54] use a simulated architecture in which the L1 and L2 caches are modified to tag cache lines with "WatchFlags" indicating memory addresses to be watched. Large ranges of memory addresses to be watched are also stored in an added buffer referred to as a "Range Watch Table". The reorder buffer of the processor is augmented with a "Trigger" bit, as well as each load-store queue entry with "WatchFlag" information. As a triggering load or store reaches the head of the reorder buffer, a monitoring function is invoked. All of these require extra hardware or specialized/simulated hardware in order to extract memory events and may not be practical when used on commodity systems.

2.1.3 Binary Instrumentation

Another approach used to intercept memory events is dynamic binary instrumentation (DBI). This is the act of instrumenting (adding analysis code to) a program at runtime. The inserted code allows information about the program to be gathered but does not interfere with the normal functioning of the program. In contrast to the previous section on extracting memory events from hardware, DBI requires no modifications or additions to the hardware allowing DBI tools to function on commodity systems. Systems such as Pin [24], Valgrind [29], Dyninst [7] and DynamoRIO [42] are examples of DBI systems. Valgrind's Memcheck [33] tool uses Valgrind to check for memory errors in programs such as undefined value error detection. Dr. Memory [6] uses DynamoRIO to check for similar memory errors such as memory leaks. IBM's Rational Purify [16] tool performs the same sort of check for memory errors in Windows applications. DYBOC [35] instruments binaries to check for buffer overflow attacks and attempts to recover from the buffer overflow without terminating the program. DBI can have a high overhead associated with it: for example under Valgrind's Memcheck, programs run 20 – 30 times slower than normal [33] while DynamoRIO's Dr. Memory runs on average two times faster than Valgrind's Memcheck [6]. This overhead makes the use of these types of tools suitable for debugging purposes only.

A complement to dynamic rewriting or runtime recompilation is static rewriting or instrumentation of program binaries. Static binary instrumentation (SBI) inserts additional code and data before the execution of a process. Systems such as the ERESI project's elfsh [43], PEBIL [21], and PSI [26] are examples of SBI systems. To the best of our knowledge there are no SBI systems that have attempted memory interception.

2.1.4 Emulation and Virtualization

Approaches like emulation and virtualization are common solutions for providing an environment for data collection and analysis of program or guest behaviour. Popular platforms for such work include Bochs [4] and QEMU [45] because they offer a way to easily modify CPU behaviour in novel ways. For example, Bochs has the ability to emulate different x86 CPU models ranging from Intel's 386 to x86_64 processors as well as AMD processors. It also has the ability to emulate a generic CPU type which allows the user to specify settings in the CPU [40]. Supervision environments and virtualization also offer the ability to support time-travel debugging [19] because of snapshot facilities included in the platform. Time-travel debugging is the ability to debug either forwards (as a regular debugger does) or backwards in time to replay crucial moments in the execution of a program such as an operating system.

Bochspwn [18] instrumented Bochs in order to spot time-of-check-to-time-of-use (more generally known as double-fetch) vulnerabilities in the Windows kernel. Interestingly, they mention the approach Cage uses as a potential design option but dismiss it in favour of using Bochs, the "simplest to quickly implement" [18, page 17]. Hobbes [8] uses an interpreter along with shadow memory to perform run-time type checking for memory and register locations. While Hobbes does not use an emulator like Bochs or QEMU, it does use an interpreter to interpret each instruction before that instruction is executed. Because emulation and virtualization can introduce significant performance limitations, they suffer drawbacks similar to DBI when used for this type of memory interception.

2.1.5 Reuse of Hardware Facilities

Most interception techniques need to rely on some physical component of the system in order to function reliably. Hardware debug registers provide an efficient mechanism

for "watching" a small number of memory locations and can even be used for stealthy supervision [15]. Unfortunately, given the small number of these debug registers available (only four are available on Intel processors [17]), watching a large set of addresses is infeasible.

Overloading the bits in the Page Table Entry (PTE) to trigger a page fault is another common approach and has been used or proposed by a variety of projects [36, 39, 37, 2, 30, 28, 3] and will be discussed in more detail in the following section. Finally, despite its utility, few systems make use of the hardware memory segmentation and the overloading of the Descriptor Privilege Level (DPL) bits. SegSlice [5] overloads the DPL bits to force a trap to the operating system on any memory access. This trap occurs because the privilege level represented by the DPL bits is incorrect, resulting in a privilege violation. Memalyze [36] proposes the use of the null selector loaded into the segment registers to generate a general protection fault on each instruction that executes a memory access. This general protection fault can then be handled specially by the operating system. Vx32 [13] uses segmentation to provide sandboxing of a process' data. The data sections for a process are placed in a separate segment and any accesses outside of this segment will produce a segmentation violation. While not concerned with intercepting memory accesses, Vx32 is nevertheless an example of using segmentation to monitor memory accesses. Unfortunately, segmentation is rarely used to its full potential in commodity operating systems, such as the Linux OS we chose to work with, resulting in its deprecation as seen in the Intel Documentation.

> "In 64-bit mode, segmentation is generally (but not completely) disabled, creating a flat 64-bit linear-address space. The processor treats the segment base of CS, DS, ES, SS as zero, creating a linear address that is equal to the

> effective address. The exceptions are the FS and GS segments, whose segment registers (which hold the segment base) can be used as additional base registers in some linear address calculations." [17, Section 3.7.4.1]

2.2 Page Fault Mechanism

Cage makes use of existing hardware facilities. This approach was chosen because of its ability to function on commodity systems without the need to modify a program binary or emulate an execution environment. Specifically, Cage overloads the User/Supervisor bit in a page table entry (PTE) to generate a page fault when the page corresponding to the PTE is accessed. This page fault mechanism has been proposed and used in previous work such as Memalyze [36], OllyBonE [37], grsecurity's PAGEEXEC [39], ELFbac [2] and the Linux kernel's own kmemcheck [30] and mmiotrace [28]. Indeed, the Page Fault Weird Machine [3] demonstrates how much power lurks in the paging circuitry.

Memalyze [36] proposes the page fault mechanism as a way to trap memory events and also proposes the use of mirrored page tables as a way to allow the user mode access to memory without the need to reset the bit in the PTE. Cage implements a variation of the proposed page fault mechanism but does not implement mirrored page tables. (Cage does not need mirrored page tables because it resets the bit in the PTE before allowing the instruction to execute.) The grsecurity team's PAGEEXEC [39] used the page fault mechanism to implement non-executable pages in Linux before the execute disable (XD) bit was added to the PTE. The Linux kernel's kmemcheck [30] uses the page fault mechanism to debug kernel code for memory errors much in the same way as Valgrind's Memcheck does for user level processes. The mmiotrace [28] tool is used to debug memory-mapped IO and was created for use with the Nouveau

project [44] (an open source Linux video driver for nVidia cards) in mind. Cage's page fault mechanism is inspired by the design proposed in Memalyze and the implementation done by grsecurity's PAGEEXEC and the Linux kernel's kmemcheck and mmiotrace.

OllyBonE [37] is a plugin for OllyDbg [51]. The two work together to produce a semi-automatic tool for unpacking malware. OllyBonE uses a similar method to grsecurity's PAGEEXEC to set pages as non-executable. This causes the first attempt to execute a code section to return control to OllyDbg. The code section to watch is specified through OllyDbg and contains the unpacked code. While Cage does have the ability to intercept memory events on a page level granularity, through the added `chmem` system call, it focuses mainly on intercepting the memory events for an entire process. Cage can also intercept both instruction execute and data access memory events while OllyBonE focuses only on the instruction executions.

ELFbac [2] describes how to combine ELF section names, using a similar page fault mechanism, to trap certain code-data ownership relationships. ELFbac focuses primarily on the access control and labelling scheme behind marking pages. While Cage does have the ability to label pages it does not yet implement access control schemes built on these page labels.

2.3 Finding Patterns in the Memory Event Stream

Much of the work discussed above intercepts memory events with the goal of watching the memory event stream for patterns. These patterns can express memory bugs or errors in the program such as memory leaks, memory corruption, and buffer overflows and can be used for debugging purposes. Many of these memory errors are indicative of vulnerabilities in the program which is demonstrated by [18, 53, 54, 33] which use programs with known vulnerabilities as tests to evaluate the effectiveness of the

mechanisms proposed.

MemTracker [50] uses a memory interception mechanism to watch for heap buffer overflows from sequential accesses, modifications of return addresses on the stack, and reads from uninitialized heap data. The above is achieved by maintaining three bits of state for each word of memory corresponding to uninitialized, initialized and unallocated data. The memory allocation and deallocation libraries are modified separately to implement the three types of patterns and set the state bits accordingly. Patterns are identified/enforced during execution of a program by updating the state bits accordingly and ensuring that no memory access occurs to an invalid state. For example, reads from uninitialized heap data are noticed by observing a memory read to a word of memory whose state is uninitialized.

iWatcher [54] tests programs for memory leaks, memory corruption, buffer overflow, stack-smashing attacks, value invariant violations and out-of-bound pointers. The above is achieved by using customized monitoring functions for each different type of pattern. (It is unclear what computational power these monitoring functions contain.) Each memory access to a watched location is referred to as a "triggering access" and causes the monitoring function specified for that range of memory addresses to be invoked. For example, to detect buffer overflows, padding is added to all buffers and the addresses of the padding are monitored. Any access to the addresses of the padding is considered an instance of a buffer overflow.

Valgrind's Memcheck [33], IBM's Rational Purify [16], and DynamoRIO's Dr. Memory [6] check for similar memory errors as iWatcher. Valgrind achieves this by associating a "V" or "valid-value" bit with each bit in memory and an "A" or "valid-address" bit with every byte of memory. The A bits are checked on every memory access to determine if that location in memory is valid to access at that particular time. For example, all the bytes in an allocated array are marked as accessible but

any access beyond the end of the array accesses memory marked as inaccessible. The V bits track whether or not a bit in memory is initialized. When a value in a CPU register is a memory address or is used in a conditional branch statement the V bits are checked to ensure that all of them are initialized.

AccMon [53] uses iWatcher to find and enforce what it refers to as "Program Counter-Based Invariants", which are the set of instructions that access a particular memory location, referred to as an "AccSet". This AccSet is designed to represent the relationship between a memory address and the set of instructions that access it. A training run, on bug-free input, is used to establish the AccSets of the program. An enforcement run uses the established AccSets to detect instructions that access a memory address but are not contained in the AccSet. These types of accesses are indicative of memory errors such as buffer overflows, stack-smashing attacks, dangling pointers and memory corruption. Indeed, AccMon is shown to be able to successfully detect these types of errors in programs.

Hobbes [8] identifies two types of patterns which are referred to as "access errors" and "type errors." A memory access error is defined as a read to an invalid memory location such as accessing an unallocated or uninitialized memory address. A type error is defined as an operation performed on operands whose types are not compatible such as adding a real number to a pointer, dereferencing an integer, and calling a function with the wrong number or type of arguments. The types represented are primitive C language types and the type information is extracted from symbol and debug tables in a program binary. Each byte of memory is associated with its type represented as a byte in shadow memory. These types are updated by the memory management library routines which have been modified. When an instruction is interpreted by the interpreter, the types of the source and destination memory locations are checked to ensure that the instruction being executed can be performed

with those operands.

Bochspwn [18] focuses on checking for time-of-check-to-time-of-use vulnerabilities in the Windows kernel. It has both an offline and online analysis mode. In the offline analysis mode, information about every instruction is stored in a separate log file that can then be processed later to find patterns. In the online analysis mode, the time-of-check-to-time-of-use pattern specified is identified during the runtime of the system. Other online patterns may be specified, but they would have to be built into Bochspwn itself, requiring modifications and additional runtime overhead. Bochspwn identifies other patterns of interest but leaves the implementation of searching for these patterns as future work.

MUVI (MUlti-Variable Inconsistency) [34] focuses on checking for multi-variable related inconsistent updates and concurrency bugs. MUVI performs static analysis of program source code and uses data mining techniques to detect variables that are consistently updated together, and are therefore correlated, with the goal of detecting instances where these variables are not updated together. In addition, MUVI also checks for multi-variable concurrency bugs by ensuring that the same lock covers accesses to correlated variables and that the ordering of the accesses is consistent indicating that no race condition exists.

All of the above work focuses on detecting memory errors by searching for pre-defined patterns. The majority of the work requires that the patterns be specified as a part of the mechanism. Any modification or additions to the patterns requires the mechanism to be recompiled at the least. Cage allows the patterns to be specified separately from the mechanism and so allows new patterns to be specified without the need to modify the mechanism. Cage also has the ability to make a run-time decision regarding a memory access based on the pattern specified. This means that during the execution of a program Cage can evaluate each memory access against the

Related Work	Speed	Full Transparency	Light Transparency	Reliability	Completeness
Bochspwn			X	X	X
Memcheck				X	
MemTracker	X			X	X
iWatcher				X	X
Cage			X	X	X

Figure 2.1: A comparison of the work most closely related to Cage in terms of its filtering capabilities.

specified filter, allowing Cage to decide whether a particular memory access matches the pattern specified. If the memory access violates the pattern, Cage can raise a segmentation fault to the running program indicating that a memory violation has occurred and that the program should be terminated. Cage also has the ability not only to intercept the memory event stream but also to modify the event stream. While other work also has this ability, the focus of the other work has been on observing the memory event stream with the goal of detecting specific patterns.

Based on the above work we have selected four of the most closely related pieces of work in terms of the filtering aspect, and compared these based on the principles we defined in Chapter 1. Figure 2.1 shows our interpretation of this comparison based on the information available from the referenced papers. Only Bochspwn [18] and Cage meet the lightly transparent principle as neither require modifications to the source code or binary image of the target program. We consider binary instrumentation to be a modification of the binary and so do not count this as adhering to the lightly transparent principle. We also do not count binary instrumentation to be complete as it cannot intercept and filter each instruction level memory event generated by the target process. While both MemTracker [50] and iWatcher [54] are hardware level solutions we believe that only MemTracker achieves the principle of speed. While MemTracker does have some amount of overhead, this overhead is relatively small in comparison with iWatcher. From this comparison it is clear that Cage is most closely comparable to Bochspwn based on the principles achieved by each mechanism.

2.4 Summary

In this chapter, related work was discussed in two parts. The first part discussed memory interception techniques and focused on the page fault mechanism that Cage uses. Memory interception can be done using a variety of methods including standard or "built-in" tools such as gdb [12], hardware additions or modifications such as those used in CoPILOT [31] and MemTracker [50], dynamic binary instrumentation such as Valgrind's Memcheck [33], emulation or virtualization uses such as Bochspwn [18], and reusing existing hardware features such as the page fault mechanism that Cage uses.

The second part focused on the ability to find patterns in the memory event stream. While there are many different techniques used to find patterns in the memory event stream, the commonality across all of the work is the motivation behind it. The goal behind all of the work was to detect varying types of memory errors such as memory leaks, memory corruption, buffer overflows, stack-smashing attacks, value invariant violations, out of bound pointers, and type errors.

In general, as demonstrated by the related work discussed here, there is no solution to intercepting memory events that is readily available on commodity systems. All of the techniques discussed involve using an additional tool or functionality of the system to achieve memory interception. In some cases, the techniques used are an abuse of pre-existing hardware features demonstrating that the hardware is capable of this type of memory interception. Unfortunately, as demonstrated by techniques using hardware additions or modifications, it is expensive, computationally and financially, to introduce an additional set of hardware primitives tailored to intercepting memory events. However, introducing a commodity mechanism for intercepting memory events would allow further work to be accomplished in detecting patterns in the memory event stream. Recognizing patterns in the memory event stream is a non-

trivial task and often involves maintaining a significant amount of state. However, recognizing patterns in the memory event stream has been shown to be useful in detecting memory errors such as memory leaks, memory corruption, buffer overflows, stack-smashing attacks, value invariant violations, out of bound pointers, and type errors. The next chapter discusses the main Cage mechanism in detail.

Chapter 3

Cage Mechanism Design

In order to intercept the memory events of a process, Cage uses a page fault mechanism that consists of a series of modifications to the memory management subsystem of the Linux kernel. This page fault mechanism triggers a page fault given any attempt by a "caged" user-level process to access its process address space. ("Caged" is used to refer to any page or process that is being supervised by the Cage mechanism.) The steps outlined below are an overview of the steps the Cage mechanism takes. Figure 3.1 illustrates the flow of the Cage mechanism and begins at step 2.

1. Mark vm_areas and all pages within a vm_area as caged (*3.1*)
2. Trigger a page fault by a user-level access to a caged page (*3.1*)
3. Ensure that the page fault was triggered by the Cage mechanism (*3.1*)
4. Evaluate BPF filter over the memory event (*3.4*)
5. Generate memory event packet for the memory event (*3.2 and 3.3*)
6. Mark the page as un-caged (*3.1*)
7. Store the faulting addresses (*3.1*)
8. Enter single step mode, disable interrupts, and restart instruction (*3.1*)
9. Trigger a debug fault at the beginning of the next instruction (*3.1*)
10. Mark the page as re-caged (*3.1*)
11. Flush the entries for the faulting addresses out of the TLB (*3.1*)
12. Exit single step mode, enable interrupts, and continue execution (*3.1*)

The discussion of the steps outlined above are divided into the following four major sections, as indicated by the italicized section numbers above. Steps 1-3 and 6-12 are discussed in Section 3.1, which discusses the page fault mechanism. Step 5

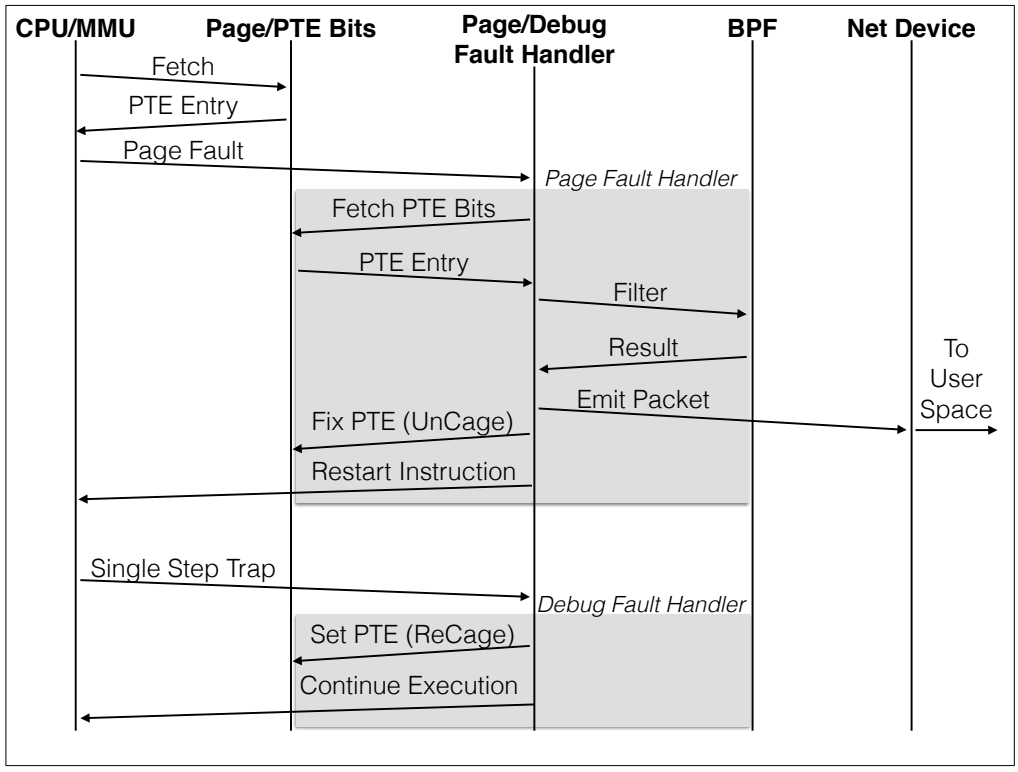

Figure 3.1: Flow Diagram of the Cage Mechanism Workflow.

is discussed in Sections 3.2 and 3.3. Section 3.2 discusses the memory event packet format and the creation of these memory event packets while Section 3.3 discusses the memory network device and the Wireshark [47] dissector that views the memory event packets. Step 4 is discussed in Section 3.4, which discusses the modifications that we made to the Linux implementation of the Berkeley Packet Filter (BPF) [25] and the filter mechanism that Cage uses. Section 3.5 describes a working example of the operation of Cage. The remaining sections discuss the supporting mechanisms that have been created including varying mechanisms to cage different vm_areas as well as special memory event packets showing the creation of vm_areas.

3.1 Page Fault Mechanism

In addition to the modifications we made to the Linux kernel's memory management subsystem, the page fault handler of the Linux kernel has also been modified with Cage-specific code to correctly handle page faults resulting from the Cage mechanism. This section details the steps involved in the Cage mechanism's memory interception technique.

Step 1: Mark vm_areas and all pages within a vm_area as caged

Virtual memory areas (vm_areas) are caged by modifying the page protection bits of the vm_area to be in a unique state. The `vm_area_struct` is a C structure that contains metadata about a vm_area including the `vm_page_prot` field. The `vm_page_prot` field is a 64-bit field containing information about the page protections such as read, write, and execute permissions. Figure 3.2 shows the layout of the bits in the `vm_page_prot` field as well as in the page table entry (PTE) for a page. The `vm_page_prot` field also contains a bit indicating whether the pages are user or supervisor pages (U/S bit) (user pages are user-level pages whereas supervisor pages are kernel-level pages). Pages inherit the information contained in the `vm_page_prot` field in their PTE. Thus, any new page created in a vm_area will have the information from the `vm_page_prot` field in their PTE and will be marked as caged without requiring further intervention. In order to mark a vm_area as caged, the U/S bit is cleared indicating that the pages will be supervisor pages. Any access to this vm_area by the owner user-level process will trigger a page fault because it is an attempt by a user-level process to access a kernel-level page. In addition to clearing the U/S bit, caged pages are also identified by setting bit nine (Cage bit). This bit is labelled as being unused (it actually is used to indicate a special mapping that should not be associated with a `struct page`). It is the combination of the U/S bit being cleared and the Cage bit being set that indicates a caged vm_area or page. In addition to

63	62...52	51...12	11...9	8...3	2	1	0
XD	ignored *Cage label*	physical frame address (high bits reserved)	ignored *Cage bit 9*	status bits	U/S	R/W	P

Figure 3.2: PTE diagram, based on [17]. Cage reserves bits 52–62 as a "label". Like previous approaches to such trapping, it overloads the User/Supervisor bit (2). We also overload bit 9 (to help distinguish a "Cage" page from other uses of bit 2), which is supposedly unused (but is actually used by the kernel).

modifying the U/S bit and the Cage bit of the `vm_page_prot` field, bits 52 – 62 are unused and provide the opportunity to label a vm_area or a page with an 11 bit long identifier. While we have explored the ability to label vm_areas and pages, Cage does not make use of these labels at this time.

Step 2: Trigger a page fault by a user-level access to a caged page

Page faults that are triggered by Cage occur in two cases. In the first case, the page fault is the result of an instruction fetch to a code page. An instruction fetch is an attempt to read the opcode for the next instruction to be executed from a page containing code. The read event in a caged code page is what triggers the page fault. In the second case, the page fault is the result of an attempt to read or write from/to a data page. A read or write access to a data page occurs when data used by a process is either read or updated. For example, an update to a global variable within a process creates a write event to a data page. A read or write event in a caged data page triggers a page fault. For the purposes of Cage, any page that is marked as executable is considered a code page and any page that is marked as non-executable is considered a data page. This means that vm_areas such as the heap and the stack are considered to contain data pages.[1]

Step 3: Ensure that the page fault was triggered by the Cage mechanism

Once a page is marked as caged and a page fault is triggered, the page fault handler

[1]This is assuming an x86 architecture and that code and data do not reside on the same page.

must correctly recognize the cause of this page fault as originating from the Cage mechanism and allow the Cage functions to handle the page fault. In addition, the Cage functions must also recognize page faults from a caged page that must first be handled by the regular page fault handler and return control to the page fault handler. For example, accesses to a non-present page or a copy on write must first be handled by the regular page fault handler before they can be handled by the Cage functions. In this respect the page fault handler and the Cage mechanism must work seamlessly with each other to allow page faults triggered by Cage and normal page faults to be handled without one inhibiting the functionality of the other.

Step 6: Mark the page as un-caged

Once it has been determined that the page fault is indeed caused by an access to a caged page, the next step in the page fault mechanism is to mark the page as un-caged. This is done by setting the U/S bit in the PTE for the page that triggered the page fault. An un-caged page is therefore a page in which both the U/S bit and the Cage bit in the PTE are set. Setting the U/S bit in the PTE will allow the access to the page to complete when it is restarted because it will once again be labelled as a user-level page that can be accessed by a user-level process.

Step 7: Store the faulting addresses

Before the access to a page is allowed to complete by returning from the page fault handler, the faulting address or addresses must be stored. The faulting address is the virtual memory address that was being accessed when the page fault was triggered. A distinction is made between faulting addresses that were the result of an instruction fetch and those that were the result of data accesses. The faulting addresses from both an instruction fetch and a data access must be stored for later use. The distinction between an instruction fetch and a data access is to ensure that the faulting addresses for both are stored if the same instruction causes both to occur.

In total up to six faulting addresses must be stored in some cases. These are cases in which an instruction crosses a page boundary and two faulting addresses must be stored, and/or the instruction causes either or both of the source and destination operands to cross a page boundary requiring up to four more addresses to be stored.[2]

Step 8: Enter single step mode, disable interrupts, and restart instruction

Once the faulting addresses are stored, the processor is placed in single-step mode and interrupts are disabled. The mechanism then returns from the page fault handler. This causes the processor to restart the current instruction or memory access which will succeed normally at this point.

Step 9: Trigger a debug fault at the beginning of the next instruction

The beginning of the next instruction will trigger a debug fault because the processor is in single-step mode. The debug fault handler has also been modified to detect this specific case of a debug fault and allow the Cage functions to handle it accordingly.

Step 10: Mark the page as re-caged

While in the debug fault handler, the page or pages that were un-caged in the page fault handler must be re-caged. The stored faulting addresses are used to achieve this. The PTEs for the pages containing the faulting addresses are re-caged, that is, the U/S bit is cleared as it was in the initial state.

Step 11: Flush the entries for the faulting addresses out of the TLB

At this point the faulting addresses are also flushed out of the translation lookaside buffer (TLB). This step is necessary because the restarted instruction or memory access that was allowed to occur normally will have caused an entry containing the

[2]The x86 string instructions appear in the output of Cage to be a series of identical instructions whose effective addresses differ and these instructions do not span more than one page boundary for either or both of the source and destination operands.

faulting address to be placed in the TLB. If that entry is not flushed out of the TLB, then any future access to that page will not result in a page fault and will occur normally because the entry in the TLB takes precedence. The act of flushing the entry from the TLB effectively means that for a process that has every vm_area caged the TLB is essentially unused, which forces a page table walk to occur for every memory access for that process.

Step 12: Exit single step mode, enable interrupts, and continue execution

After flushing the TLB of up to six faulting addresses the processor is removed from single-step mode and interrupts are again enabled. The mechanism then returns from the debug fault handler and the next instruction begins. If that instruction *also* accesses a caged page then the whole process is repeated beginning with another page fault (Step 2). Figure 3.1 (page 24) shows an overview of the steps necessary for the Cage mechanism to function.

3.2 Memory Event Packet Format and Generation

While in the page fault handler context, Cage is able to extract the following metadata about the memory event:

1. the instruction pointer *. . . bytes 0–7*
2. the effective address (faulting address) *. . . bytes 8–15*
3. the instruction label (Cage "label", if any, of the page containing the faulting instruction) *. . . bytes 16–17, 11 bits*
4. the effective address label (Cage "label") *. . . bytes 17–19, 11 bits*
5. the PTE meta-data for the effective address (first 10 bits of the PTE) *. . . bytes 19–20, 10 bits*

6. type of access: r/w/x ... *byte 21*
7. the instruction at the instruction pointer ... *bytes 22–36*
8. the PID and UID ... *bytes 37–40 and 41–44*
9. the data at the effective address ... *bytes 45–52*

The italicized text above indicates where the information is placed within the memory event packet structure. The memory event packet structure is a C structure that contains the above information, whose structure is shown in Figure 3.3.

Step 5: Generate memory event packet for the memory event

The above information is extracted before the page is marked as un-caged, as described in the previous section. The extraction of this information is dependent on the memory network device, described in the next section, being loaded and set to "up". If the memory network device is not loaded and "up", the packet generation will not occur and no memory event packet structures will be produced. Once the information extracted is placed in an instance of a `struct memevent_packet`, the `memevent_packet` is placed on a queue and the `net_device` struct associated with the memory network device is found. The transmit function for the memory network device is called and the page fault mechanism continues by un-caging the page (Step 6).

3.3 Memory Network Device and Wireshark Dissector

The memory network device is a Linux loadable kernel module (LKM) that creates a new network device within the operating system that can be used exclusively for memory event packets. This memory network device differs from traditional network devices in that it does not handle regular network traffic and all functionality of the device is tailored specifically to handle memory event packets. When this LKM is

```
struct memevent_packet {
          unsigned long src;
          unsigned long dest;
          unsigned int src_dest_pte;
          unsigned char rwx;
          unsigned char instruction[15];
          unsigned int pid;
          unsigned int uid;
          unsigned long data;
}
```

Figure 3.3: Structure of memory event packet.

loaded and the memory network device it creates is "up" it allocates a packet queue that is 10 `memevent_packet` structs in length. This is the same queue that the page fault mechanism places the newly formed `memevent_packet` in after it creates the `memevent_packet`.

The transmit function that is called from the page fault mechanism first dequeues the `memevent_packet` struct from the packet queue. It then encapsulates this `memevent_packet` struct in an Ethernet header. This encapsulated `memevent_packet` struct is referred to as a memory event packet. A memory event packet appears to the operating system as a normal network packet and is treated as such by the operating system. A struct `sk_buff` is created and the memory event packet is placed in the `sk_buff`. The function `netif_rx` is then called to pass the `sk_buff` to the receive network path of the kernel. (It could easily be transmitted over the network at this point but doing so is outside the scope of this work.) This arrangement effectively turns this memory network device into a loopback device.

The memory event packets can then be captured from the memory network device using any standard network packet capture utility such as Wireshark [47], dumpcap [46], or tcpdump [49]. As Wireshark provides a graphical interface that can display network packet contents and could be extended to support memory event

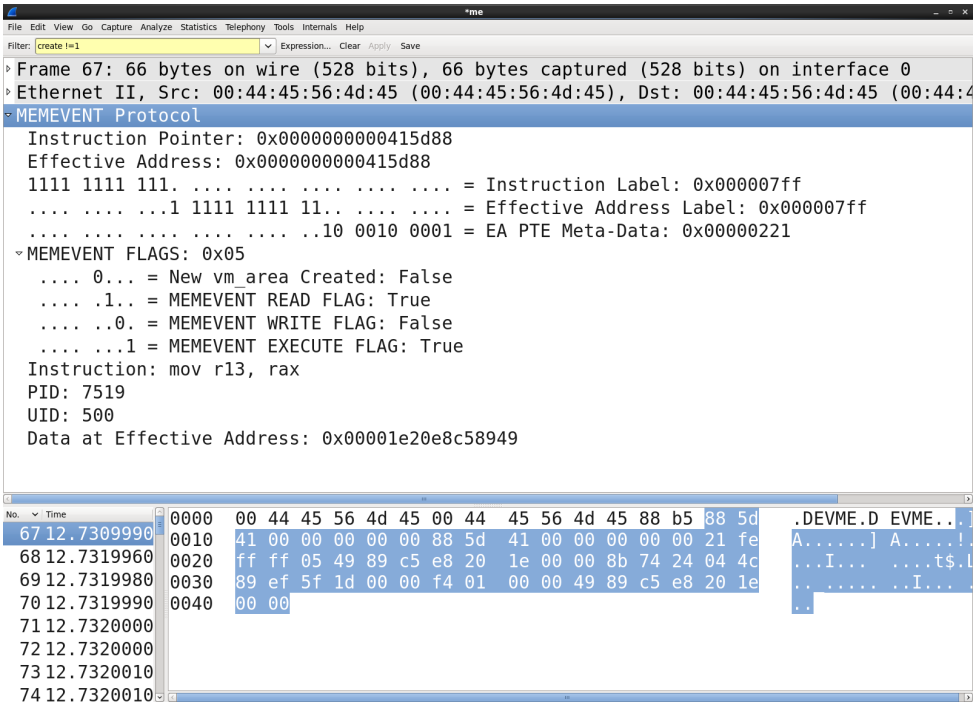

Figure 3.4: Wireshark view of an instruction fetch memory event packet.

packets, it was chosen as the primary network capture utility. To that end, a specially created Wireshark dissector is used to dissect the memory event packets so that Wireshark can display their contents. This Wireshark dissector takes the information contained in the `memevent_packet` struct shown in Figure 3.3 and displays it without modification. The only exception to this is the 15-byte instruction opcode. For the instruction opcode, the Wireshark dissector uses the libudis86 [48] library, for disassembling x86 and x86_64 instruction opcodes, to disassemble the 15-byte instruction opcode into a text string. This text string is then displayed in the packet view with the rest of the information from the memory event packet. Figure 3.4 shows an example of the Wireshark view of a memory event packet that contains information about an instruction fetch.

3.4 Filter Mechanism and Modifications to BPF

The Berkeley Packet Filter (BPF) [25] framework is a filtering mechanism for network packets that is included in the Linux kernel. Using the BPF framework to filter memory event packets allows the use of an already established filtering mechanism. In addition, the use of BPF for processing (and modifying) streams of memory events is a design alternative to the practice of inserting arbitrary C or x86 code into dynamic memory analysis. This kind of pattern has many precedents: consider the use of SystemTap built on top of kprobes. While inserting raw C or x86 code provides a great deal of control, it also entails some risk (the inserted code may be buggy or hard to maintain). In contrast, BPF offers a still–powerful interface for analysis, but avoids some of the risks of inserting arbitrary computation into the instrumented memory event sequence.

In order to use BPF to filter memory event packets and provide flexibility in the types of filters available, it was necessary to extend BPF in three ways. First, the Linux kernel implementation of BPF was originally intended to only process 32-bit values. Since Cage was designed to work on 64-bit systems it was necessary to extend the implementation of BPF to support 64-bit values. This allowed the load and store of 64-bit memory addresses. The one exception to this is the size of "K" in the BPF filters which remains a 32-bit value. K is used to pass arguments to BPF instructions in the filters and the extension of K to 64-bits for use by Cage could impair the functionality of other BPF filters.

The second modification to BPF was the addition of a load instruction that was specific to Cage. This load instruction is able to load in values about a memory event such as:

1. The instruction pointer
2. The effective address (faulting address)

3. The value contained in the rax register
4. The value of the rwx flags
5. The 8-byte value of the data contained at the effective address
6. The value contained in one of the five storage locations

This was done so that BPF filters can be created which compare these values to a given constant or range of values and allow a decision to be made based on the current memory event. Adding in the ability to load the current value in rax allows the ability to capture the return value of a function call. For example, this allows the ability to capture the address returned by a call to malloc() and store this address to allow any future memory events that involve this buffer to be noticed. This becomes important because, in some cases, the compiler could optimize the code such that the buffer address is never written to memory and is thus not viewable as a memory event.

The third modification to BPF was the addition of a store instruction that was specific to Cage. This store instruction allows the ability to store up to five different values across runs of the filter. This allows information or state that is affected by one memory event to be used to make a decision about future memory events. A more in-depth discussion about the types and capabilities of BPF filters is deferred until Chapter 5.

The BPF filters are specified in an LKM which must be loaded before the caged process being filtered is executed. This provides one global filter that is applied to any caged process that is executed. While it is possible to specify filters on a per-process basis, Cage does not attempt to do so at this time. It is assumed that the main use case for Cage will be in caging and monitoring one process at a time although there is nothing preventing multiple processes from being caged at the same time.

63	62...52	51...12	11...9	8...3	2	1	0
XD	label 11111111111	physical frame address	xx 1	status bits	0	0	1

Figure 3.5: PTE contents for the workflow example.

Step 4: Evaluate BPF filter over the memory event

The BPF filter is evaluated in the page fault mechanism of Cage before the page is un-caged and prior to the memory event packet being generated. As mentioned previously, the actions Cage performs in the page fault mechanism occur before the instruction or memory access actually occurs. Evaluating the filter at this point allows a decision to be made regarding the memory event that is about to occur. The result of this decision takes three forms. First, the BPF filter can indicate (via the return value of `sk_run_filter`) that a memory event packet should be generated and the memory event should be allowed to proceed normally. Second, the BPF filter can indicate that no memory event packet should be generated and the memory event should be allowed to proceed normally. Third, that no memory event packet should be generated and that the memory event should not be allowed to proceed. In this case a segmentation fault is raised and the executing caged process is terminated. (It is also possible to take other steps at this point such as issuing a warning to a log instead of raising a segmentation fault.) Filtering in this fashion, where the filter is evaluated on each memory event, allows a decision to be made on each memory event and effectively filters the memory event stream of a caged process as a whole.

3.5 Working Example

To illustrate Cage's workflow, consider the instruction `mov rax, [rbx+0x60014c]` as an example. The memory location 0x60014c, a constant in the .data section, has the page table entry shown in Figure 3.5. Bit nine in the page table entry is set and the

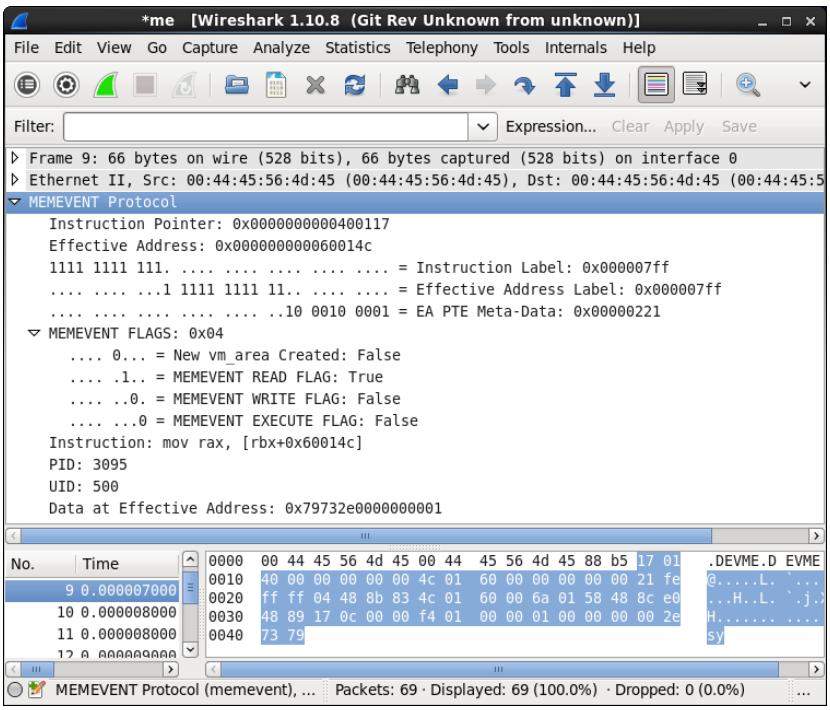

Figure 3.6: Workflow example memory event viewed in Wireshark.

user/supervisor bit (bit 2) is cleared indicating that this is a caged page. This results in a user access to a supervisor page which triggers a page fault. Once in the page fault handler the page fault is recognized as being caused by the Cage mechanism by the unique combination of PTE bits. Once in the Cage functions the BPF filter is first evaluated over this memory event. If there is no filter or if the memory event matches the filter a memory event packet is generated. This memory event packet is then sent to the network using the process described previously. The user/supervisor bit is then set in the page table entry (un-caging the page), the processor is placed in single step mode and Cage returns from the page fault handler. The execution of the instruction `mov rax, [rbx+0x60014c]` completes normally, and the beginning of the *next* instruction results in a debug trap. Here the user/supervisor bit is cleared (caging the page and returning the PTE to its starting state seen in Figure 3.5), the

processor is taken out of single step mode and the single entry corresponding to the page table entry for the address 0x60014c is flushed out of the TLB. The resulting packet generated from this instruction can be seen in Figure 3.6. Note that if the text segment containing the instruction were also caged, then two events would be seen: one for the instruction's page, followed by one for the data page.

3.6 Creating Caged Programs

Four different mechanisms exist that can be used for caging a vm_area or a process. The first of these mechanisms is a modified `mmap` system call [27] that can be used for creating a caged vm_area. The second mechanism is a new `chmem` system call that can be used to modify the caged state of the vm_areas of a running process. The third mechanism is a library that can be used in conjunction with LD_PRELOAD to cage specific vm_areas. The fourth mechanism is a wrapper program for a `clone` system call [9] that cages every vm_area in the process that is executed.

3.6.1 Instrumented `mmap` System Call

The `mmap` system call [27] allows a program to create a new vm_area within its process address space. This vm_area can be used to map in the contents of a file or can be left as an empty "anonymous" vm_area. The `mmap` system call also allows a program to specify the protections on the new vm_area, such as read, execute, and write permissions, as a set of flags that are passed to `mmap`. Cage specifies a new flag (`PROT_CAGE`) that can additionally be passed to the system call, requesting the created vm_area to be caged. This modified `mmap` system call can be used to selectively create caged vm_areas within a process. This gives the programmer the ability to selectively create a special caged vm_area which could be used to hold a critical piece of data, such as a private key. This would allow the programmer to observe any memory

events that access this piece of data.

3.6.2 The `chmem` System Call

We created the `chmem` system call to allow the ability to modify the caged state of selected vm_areas of a currently executing process. We also created a user-level utility, `chmem_user`, to act as a front end to the `chmem` system call. The vm_area to be modified is specified as a parameter to `chmem_user` and is passed as an address to the `chmem` system call. The vm_area that contains this address is found and the vm_area page protection bits are modified accordingly along with the PTEs of all the pages within that vm_area. There are six modifications that the `chmem` system call can make. The first is to cage the vm_area specified so that all future accesses to this vm_area trigger the Cage mechanism. The second is to un-cage the vm_area specified so that no future access to this vm_area will trigger the Cage mechanism. The third is to modify the label of the vm_area. The fourth is to Cage only a specified address range. This does not Cage an entire vm_area but only the pages contained in the specified range. The fifth is to Cage a single page. The sixth is to Cage only the stack. The combination of the `chmem` system call and `chmem_user` allows easy modification of the caged state of any currently executing process without requiring access or modification to the source code of the executing process. This allows a programmer to target an already executing process and cage vm_areas or pages of interest, such as the stack in order to view the memory events for the caged region of the program.

3.6.3 The Cage Library

The Cage library `libcage` is designed to only cage a specific ELF section. The `libcage` program uses the `libelf` API to find the address and size of a specified ELF section, such as the .text or .data sections. The `chmem` system call is then

used to cage the address range corresponding to the specified ELF section. The `libcage` program is a shared library that is used in conjunction with LD_PRELOAD to cage only the specified section of the program that is executed. Another version of `libcage` will cage only the stack of the executed program. This version makes use of the `chmem` system call's ability to cage the stack. The Cage library allows the ability to cage specific ELF sections while executing a program from its beginning. Unlike the `chmem` system call, it does not require the target program to be running and unlike the instrumented `mmap` system call, it does not require modification to the source code. This allows the programmer to again target a memory region of interest such as the stack or .text sections and view the memory events generated by memory accesses to that region. This also executes the target program from the beginning allowing the programmer to view all memory events for the specified region over the life time of a process.

3.6.4 Executing Caged Programs

The `cage_wrapper` utility is a wrapper for a `clone` system call. The `clone` system call creates a child process of the `cage_wrapper` program. A special flag, `CLONE_CAGE`, is passed by `cage_wrapper` to the `clone` system call. This flag tells the instrumented `clone` system call to set a flag in the `task_struct` of the child process indicating that this process is to be a caged process. We modified the ELF loader in the Linux kernel to check for the flag in the `task_struct` during the creation of the vm_areas of a process. If the caged process flag in the `task_struct` is set then each vm_area that is created for the child process is created as a caged vm_area. Thus, the child process created by the `clone` system call is created as a caged process that has every vm_area and page caged. The `cage_wrapper` executes the process given as an argument, as a caged process. This allows a programmer to execute any program as an entirely caged process from its beginning, allowing a record of the entire program's execution

to be created.

3.7 Vm_area Creation Packets

Vm_areas are created or modified at different points during the lifespan of a process. Some vm_areas, such as the heap, are created when they are first needed and expanded as required. Other vm_areas, such as the .text segment, are created by the ELF loader while the process is being loaded and before its execution begins. The ability to notice the creation or expansion of vm_areas is important because it provides a reference for correlating the effective address of a memory event with its vm_area. For example, the ability to recognize the creation of the heap allows any future access to the heap to be found in the packet trace simply by using the post-filtering functionality of Wireshark. To that end, we instrumented the vm_area management functions of the Linux kernel to produce a special memory event packet on any creation or modification to a caged vm_area. An unused bit of the rwx flags byte in the `memevent_packet` struct was used to signify the creation or modification of a vm_area. (See Figure 3.3 on page 31 for the `memevent_packet` struct.) The beginning and end addresses of the vm_area are contained in the vm_area creation packet as well as the vm_area page protection bits which indicate the caged status of the vm_area.

3.8 Summary

In this chapter the twelve steps that the Cage mechanism goes through to handle each memory event were discussed in detail. In Section 3.1, steps 1-3 and 6-12 were discussed as they relate to the steps that the Cage specific page fault handler and debug fault handler take to handle each memory event. Step five was discussed in Sections 3.2 and 3.3, which outlined the information contained in the memory event packet, the creation of the memory event packet, and the viewing of the memory

event packets in Wireshark. Step four was discussed in Section 3.4, which detailed the modifications made to BPF and the process of evaluating the BPF filter over a memory event. Section 3.5 outlined a working example demonstrating how a specific instruction would be handled. The remaining sections discussed the supporting mechanisms that Cage uses, including various mechanisms to cage the vm_areas of a user-level process and the usage of extra bits in the `memevent_packet` struct to show the creation of new vm_areas. The next chapter will discuss our evaluations of the Cage mechanism.

Chapter 4

Evaluation

This chapter discusses the evaluations that we have performed on the varying aspects of the Cage mechanism. The first section discusses the performance of the Cage mechanism using the SPEC CPU2006 suite [10] as a benchmark. The second section discusses capturing the memory event packets. The third section covers the generality of the Cage mechanism to different platforms. Finally, the fourth section details our attempt at comparing two memory event packet traces. We defer the discussion relating our results to the principles outlined in Chapter 1 until Chapter 6.

4.1 Performance of Cage

We tested the performance of Cage across various potential use cases for the mechanism as well as tests that highlight the overhead of different parts of the Cage mechanism. The test programs are the integer benchmarks from the SPEC CPU2006 suite [10].[1] The SPEC benchmarks were chosen because they provide a standard, well-recognized set of programs with which to test the functionality and reliability of Cage.

All of the tests were performed on two identical machines. A baseline control test was performed on each machine and the results are compared with the baseline of the machine the test was performed on. Each machine contains a 3.10GHz Intel i5 processor with 4 cores and 8 GB of RAM, running CentOS 6.4. Both machines run identical versions of the 3.9.4 Linux kernel modified with Cage code. All programs were tested with the "test" (smallest) inputs and the time measured was the "real"

[1]The perlbench benchmark is omitted due to mismatched libraries not allowing it to compile, while the h264ref benchmark is omitted due to observed instability.

time as reported by `/usr/bin/time`. Unless otherwise noted, each program was run once under each of the test cases due to the length of time required to run some programs. The baseline tests performed on each machine were performed on an unmodified version of the 3.9.4 Linux kernel. The baseline tests were performed three times and an average[2] of the three runs was taken as the baseline measurement.

There were five different tests performed in total. Test 1 compares the baseline with the runtime of the Cage version of the Linux kernel with none of the test programs caged. Test 2 compares the baseline with the runtime of the test programs entirely caged and the entire Cage mechanism active. Test 3 compares the baseline with the runtime of the test programs entirely caged but without the memory network device activated. This eliminates the overhead of the memory event packet creation and filtering aspects of the mechanism. Test 4 compares the runtime of the baseline with all of the test programs entirely caged and a BPF filter active that only allows memory event packets to be generated for the .data section. Test 5 compares the baseline with the runtime of only the stack of the test programs being caged.

Test 1: Baseline kernel as compared with Cage kernel

The first test, comparing the baseline kernel with the Cage version of the Linux kernel with none of the programs caged, is designed to test the overhead that the Cage mechanism has when it is not active. Even while inactive, there are still extra lines of code that are executed during each page fault and memory allocation that occur. This test was performed three times and the average of the runs was compared with the baseline. The average of the runs as well as the standard deviations for each of the test programs are shown in Figure 4.1. In the majority of cases the runtime of the baseline, or standard, kernel is within one standard deviation of the runtime of the Cage kernel. This indicates that there is not a significant difference between the runtime of the

[2] "average" as used throughout this thesis refers to the arithmetic mean.

SPEC Benchmark	Standard Kernel	Std Dev	Cage Kernel	Std Dev
bzip	4.84	0.10	4.77	0.06
gcc	1.07	0.11	1.08	0.10
mcf	2.23	0.02	2.07	0.01
gobmk	15.35	0.29	14.99	0.47
hmmer	2.32	0.04	2.33	0.02
sjeng	3.25	0.23	3.19	0.02
libquantum	0.03	0.01	0.04	0.01
omnetpp	0.33	0.03	0.33	0.03
astar	7.88	0.03	7.86	0.05
xalancbmk	0.13	0.12	0.12	0.10

Figure 4.1: Average run time in seconds over three runs for standard kernel and Cage kernel with no caged program as well as standard deviations for both.

baseline kernel and the runtime of the Cage kernel. (The anomalous behaviour of mcf is examined in more detail later on.) The performance of the Cage kernel as compared with the baseline kernel can be seen in Figure 4.2. This comparison highlights how close the runtime of the baseline kernel and the Cage kernel are to each other as evidenced by the values for each program closely approaching 1.0. We examined the two outlier programs, mcf at 0.93 and libquantum at 1.09, in more detail as shown in Figure 4.3. Both mcf and libquantum were run a total of 12 times and the mean and standard deviation of each is reported. With regards to libquantum, the baseline and the Cage kernel both have an average runtime of 0.04 seconds indicating that there is a negligible difference between the two cases. However, the runtime of mcf shows a significant improvement when run on the Cage kernel as compared with the baseline. On the Cage kernel mcf runs on average for 2.06 seconds while the baseline runs for 2.24 seconds. The mcf program is designed to simulate single-depot vehicle scheduling in public mass transportation [23]. We conjecture that the improvement seen in the runtime of mcf may be due to a possible cache alignment created by the Cage kernel. Part of our future work will be to investigate this further. Overall, with the exception of mcf, there is not a significant difference in the runtime of the SPEC programs between the baseline kernel and the Cage kernel.

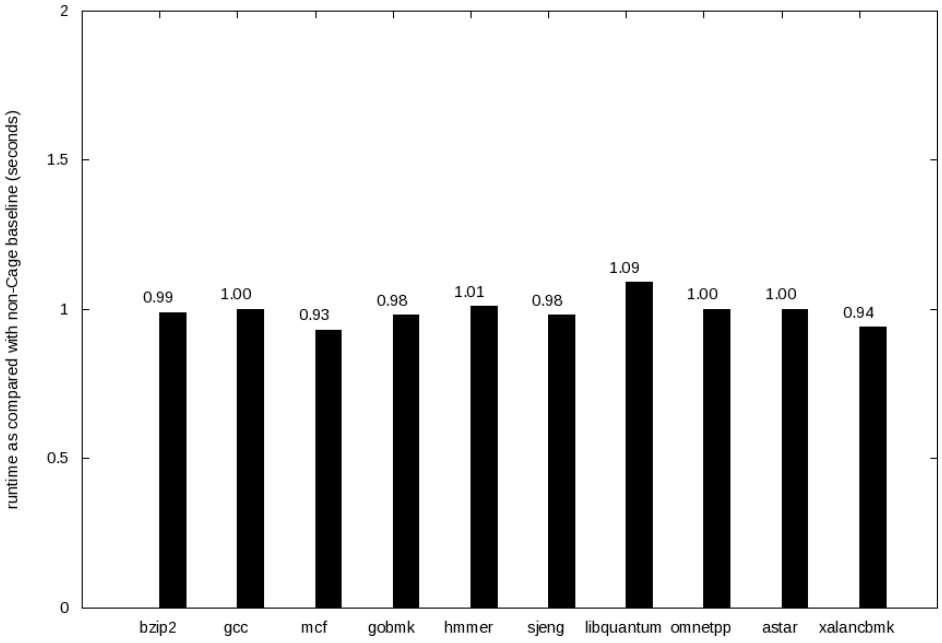

Figure 4.2: Programs are not caged, as compared to baseline SPEC runs

Test 2: Baseline as compared with 100% Caging

The second test, comparing the baseline with the entirely caged version of the SPEC programs, is designed to show the worst case scenario in terms of performance. We expect caging 100% of the process address space to have a large impact on the performance of a program. When a program is 100% caged, there is effectively no TLB because a TLB flush is performed for every memory access, as previously discussed. Also, instruction prefetching is effectively disabled because a prefetch is not able to handle the page faults. Handling a page fault on every memory access is also extremely intensive and we expect this to produce a large overhead. The SPEC benchmarks are also known to be extremely memory intensive [1]. We would expect less memory intensive programs to perform better than the SPEC benchmarks. Figure 4.4 shows the performance in seconds of the SPEC programs as compared with the baseline. The performance ranges from the fastest program, mcf at 2540 seconds slower (about 42 minutes), to the longest running program which is hmmer at

Run #	Standard libquantum	Cage libquantum	Standard mcf	Cage mcf
1	0.04	0.04	2.25	2.06
2	0.05	0.04	2.23	2.08
3	0.03	0.03	2.22	2.06
4	0.04	0.03	2.27	2.06
5	0.03	0.04	2.25	2.06
6	0.04	0.04	2.23	2.06
7	0.03	0.04	2.26	2.06
8	0.04	0.04	2.23	2.05
9	0.03	0.04	2.23	2.06
10	0.03	0.04	2.23	2.05
11	0.03	0.04	2.24	2.07
12	0.04	0.04	2.23	2.05
Mean	0.04	0.04	2.24	2.06
Std Dev	0.004	0.001	0.02	0.01

Figure 4.3: Run time in seconds of the libquantum and mcf benchmarks across 12 runs on the standard kernel and the Cage kernel with no programs caged. The mean and standard deviation of both are reported.

12454 seconds slower (about 207 minutes). We expect this rather significant overhead based on the reasons outlined above. The range of values seen is an indication of how memory intensive a particular program is. In general, the more memory intensive the program, the longer the runtime will be when the program is caged. Figure 4.5, column two, shows the results for each program as compared with the baseline.

Test 3: Baseline as compared with 100% Caging but no memory network device

In the third test, all of the programs are 100% caged but there is no memory network device installed. This results in a page fault being generated for every memory access but there are no memory event packets created. While not a probable use case for the Cage mechanism, this test is designed to investigate the overhead required to generate the memory event packets. In addition, this test also investigates the overhead produced from just handling a page fault on every memory access. The black bars in Figure 4.6 show the results of this test. The y-axis scale is kept the same as the scale from the previous test in order to compare the difference in the performance of the two tests. In general, the performance of having no memory network device

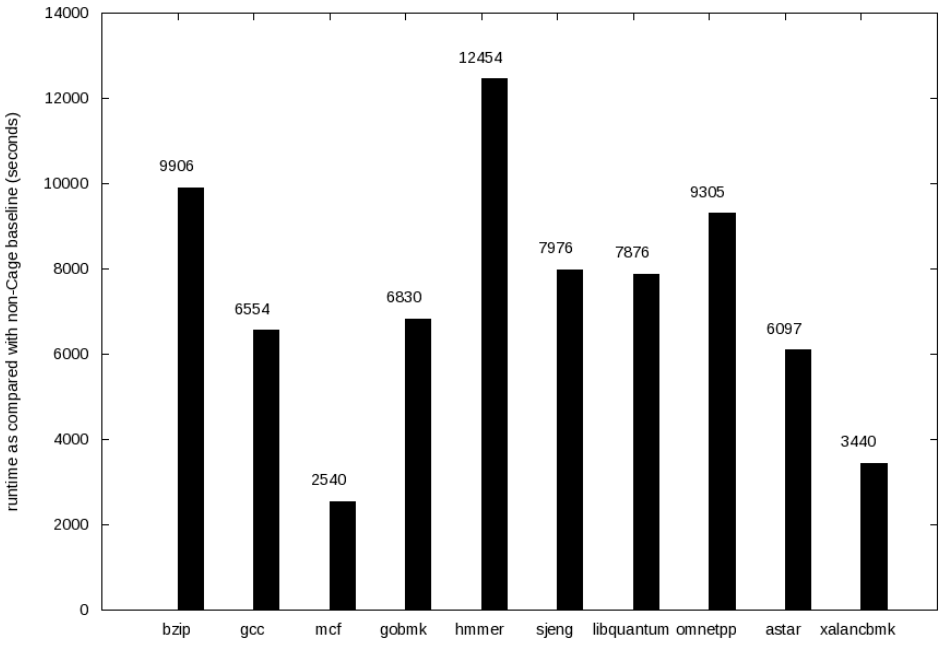

Figure 4.4: Programs are 100% caged, as compared with baseline SPEC runs (lower is better)

installed is about 50% – 60% of the performance of having the programs 100% caged. This indicates that the overhead of producing the memory event packets is almost as great as the overhead of handling a page fault for every memory event. Figure 4.5, column three, shows the results of this test.

Test 4: Baseline as compared with 100% Caging but with a BPF filter for the .data section

In the fourth test, all of the programs are 100% caged, the memory network device is active and there is a BPF filter that is only allowing memory event packets to be created for the .data section of the benchmark programs. This test is designed to show the difference in the performance of having no memory network device installed, as in the third test, and the performance of having a BPF filter for only one section of memory. This test also highlights how effective using a BPF filter can be to reduce the overhead that the second test showed while 100% caged. The grey bars

SPEC Benchmark	100% Caged	No Network Device	BPF Filter	Stack Only
bzip	9906.99	5209.93	5327.88	882.18
gcc	6554.43	3806.10	3919.45	993.14
mcf	2540.24	1302.77	1329.22	59.22
gobmk	6830.52	3447.02	3575.00	965.11
hmmer	12454.61	6194.52	6393.37	1350.07
sjeng	7976.68	4038.66	4139.52	853.57
libquantum	10063.81	5365.46	5482.61	189.69
omnetpp	9305.03	4922.89	5102.30	1507.48
astar	6097.69	3026.51	3104.48	660.26
xalancbmk	3440.90	2590.56	2680.56	368.89

Figure 4.5: Runtime in seconds as compared to baseline of standard kernel for 100% Caged, 100% Caged but no memory network device, 100% Caged with a BPF filter for .data section, and only Stack Caged.

in Figure 4.6 show the results of this test. In general, there is not a significant increase in runtime between test three, with no memory network device, and this test. Figure 4.7 highlights the difference between having a BPF filter for the .data section and having no memory network device. In the worst case, having a BPF filter for the .data section is only 1.04 times worse than having no memory network device. This is an expected result because the BPF filter limits the memory event packets that are being created to those corresponding to the .data section only. Since this is a relatively small section of memory when compared with the entire process address space, we expect that producing memory event packets for it will only incur a slightly larger overhead than producing no memory event packets. In general, the more restrictive the BPF filter, the lower the performance overhead will be. This test also demonstrates the usefulness of using a BPF filter to narrow the range of memory that will produce memory event packets when compared with the results of the second test. Once again, the performance of having a BPF filter for the .data section is about 50% – 60% of the performance of having the programs 100% caged, as in the second test. The fourth column of Figure 4.5 shows the results of this test.

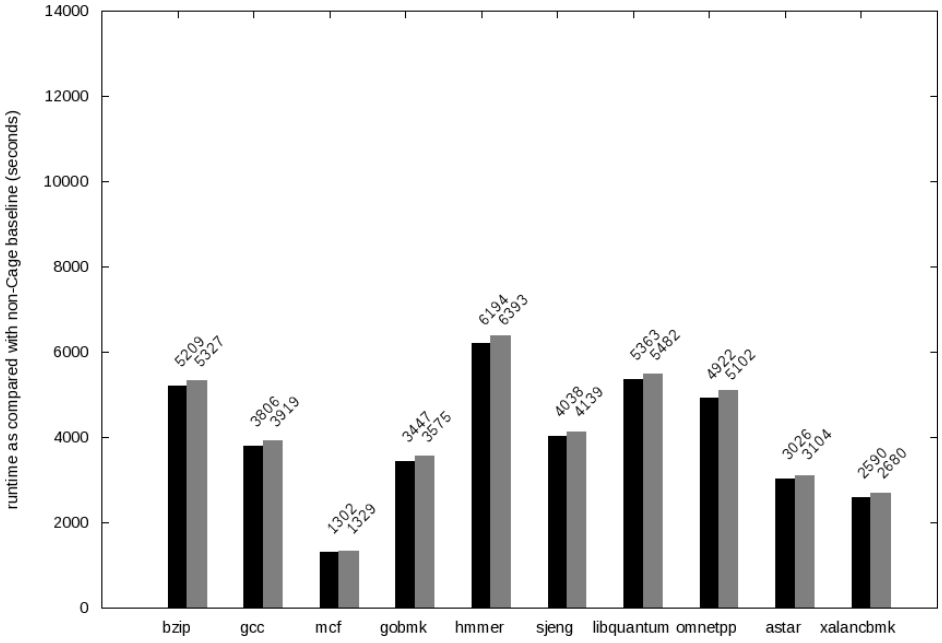

Figure 4.6: The black lines show programs that are 100% Caged but there is no memory network device, as compared to baseline. The grey lines show programs that are 100% Caged with a BPF filter that only produces packets for the .data section, as compared to baseline SPEC runs (lower is better)

Test 5: Baseline as compared with caging only the stack

In this test the stack version of the Cage library, `libcage` (discussed in Section 3.6.3, page 38) is used to cage only the stack of the benchmark programs. This test is designed to demonstrate the performance benefit of only caging the ranges of memory that one is interested in watching. We used `libcage` to perform this test as it provides a way of easily caging the stack that is consistent across all of the programs in the benchmark. Figure 4.8 shows the results of this test. The y-axis is kept on the same scale as that of Figure 4.4, showing 100% caging, so that the results may be compared. In general, caging only the stack shows a significant performance benefit. In the best case, caging only the stack has a runtime that is 2.33% of the runtime of caging 100% of the process address space. In the worst case, the runtime is 16.20% of the runtime of caging 100% of the process address space. Figure 4.9 shows the runtime

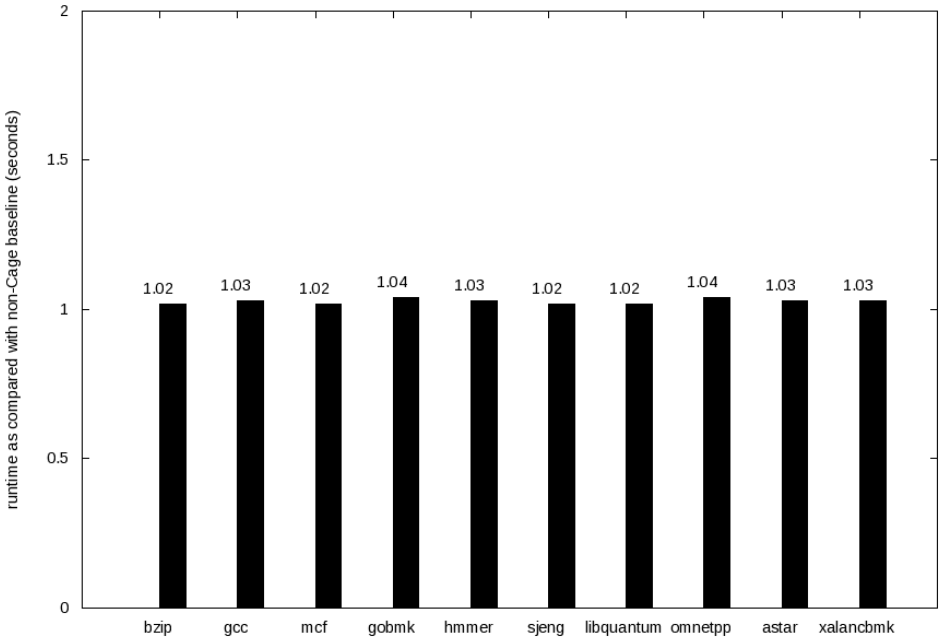

Figure 4.7: The run time in seconds for the BPF filter for the .data section as compared with the runtime in seconds of having no memory network device.

of caging the stack only as a percentage of caging 100% of the process address space. As expected, the amount of stack utilization plays a large role in the performance of this test. This explains the widely varying results in Figure 4.9. The fifth column of Figure 4.5 shows the results of this test.

While testing `libcage's` ability to cage an address range we noticed that there was a mismatch between the state of the pages as reported by the kernel versus the user-level program. Since `libcage` is loaded and executed early on in the loading of a program it was attempting to cage the pages in the address range specified before the kernel had allocated those pages. This is a form of lazy allocation that the kernel performs where it only allocates pages when they are requested by the user-level program. To fix this issue we had to access all of the pages in the range we want to cage from `libcage` by either reading from them or using `mlock` to lock the pages in the range into memory while they are being caged. This caused the kernel to allocate

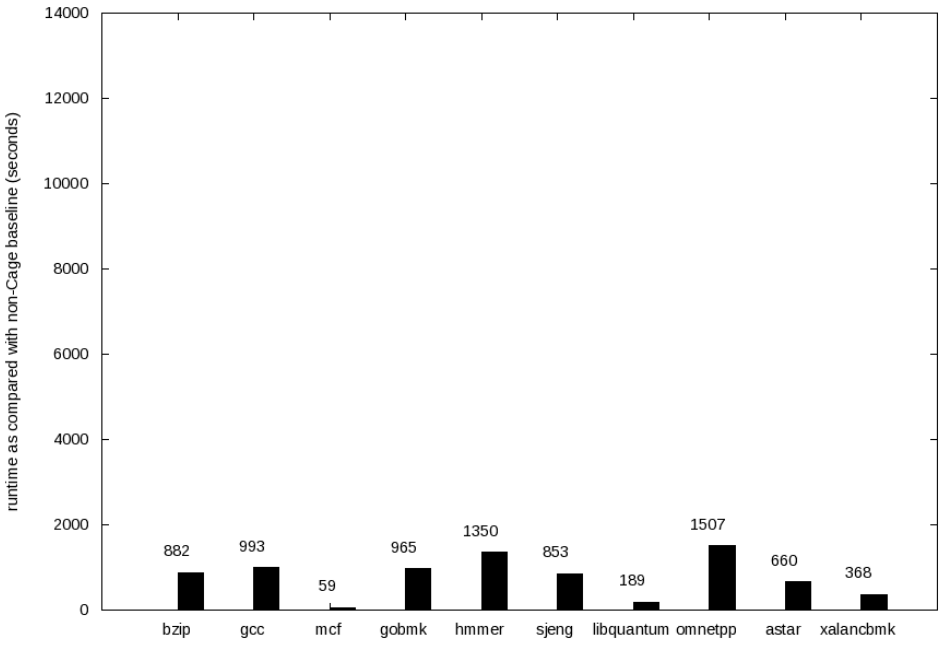

Figure 4.8: Only the stack is caged, as compared to baseline SPEC runs (lower is better)

the pages so that they could be caged. We do not test the performance of caging a range of memory using `libcage` as we believe that its performance would be similar to that of caging only the stack.

4.2 Capturing Memory Event Packets

In attempting to capture 100% of the memory event packets of a caged process using Wireshark, we realized some limitations in Wireshark's capturing capability. Wireshark, by default, maintains a small pcap buffer (2MB) to buffer un-processed packets. This pcap buffer size is not large enough to accommodate the volume of the memory event stream. Even increasing this pcap buffer size to be 200 – 500 MB was insufficient to capture 100% of the memory event packets. Wireshark also performs some analysis on the packets as it receives them, further inhibiting its capture speed. Finally, the disk I/O buffer is a fixed size and the disk I/O also contributes to inhibiting

SPEC Benchmark	Percentage of 100% Caged result
bzip	8.90
gcc	15.15
mcf	2.33
gobmk	14.13
hmmer	10.84
sjeng	10.70
libquantum	1.88
omnetpp	16.20
astar	10.83
xalancbmk	10.72

Figure 4.9: Caging the stack only runtime as a percentage of 100% caging.

the capture speed. To that end, we created a special bare-bones packet capture utility in an attempt to capture 100% of the memory event packets of a caged process. In contrast to Wireshark's default, this capture utility includes a 200 MB pcap buffer that can also be modified as needed. In addition, there is no processing performed on the captured packets, which are placed immediately on the disk I/O buffer. The disk I/O buffer is modifiable and by default is set to hold 5000 memory event packets. As it is the disk I/O that is the bottleneck, having a modifiable disk I/O buffer allows the ability to tailor the capture utility to the caged process by adjusting the buffer size based on the number of memory events expected. The capture utility saves the captured packets as a pcap file which can then be opened and viewed later in Wireshark to make use of the visual representation of the memory event packets given by Wireshark.

We attempted to capture the memory event packets of our SPEC benchmark programs. In doing so we noticed that the amount of disk space required to capture 100% of the memory event packets would overwhelm our test system's hard drive space. The program bzip2 alone produces over 3 billion memory event packets when only the stack is caged. This would require greater than 274GB of disk space. Longer running programs would produce more memory event packets and require larger amounts of disk space. Therefore we report the memory event packets generated from only

SPEC Benchmark	Runtime (Seconds)	Number of Packets	Number of Packets / Runtime
bzip	4,351.82	3,242,148,737	745,010.70
gcc	1,234.48	908,173,674	735,676.04
mcf	133.02	96,649,176	726,592.66
gobmk	14,378.26	10,643,471,067	740,247.45
hmmer	3,175.37	2,321,753,877	731,176.78
sjeng	2,804.82	2,078,825,279	741,161.48
libquantum	6.83	4,936,173	722,825.16
omnetpp	532.14	388,473,712	730,021.63
astar	5,227.94	3,804,618,130	727,747.23
xalancbmk	70.83	52,075,395	735,237.41

Figure 4.10: Runtime of caging only the stack along with the number of memory event packets for the SPEC benchmark programs. The third column shows the number of memory event packets produced per second when the stack is caged.

caging the stack in Figure 4.10. When compared with the runtime of the SPEC benchmark programs with only the stack caged, as seen in column two, it is clear that longer running programs, such as gobmk, produce more memory event packets. The third column of Figure 4.10 shows the number of memory event packets produced per second of execution time.

4.3 Generality of Cage Mechanism to Different x86 Platforms

Cage was tested on a variety of virtual machine software and running natively on both AMD and Intel processors. The goal of this test was to demonstrate the applicability and usability of Cage across different environments. It is assumed that the main use case for Cage will be running inside a virtual machine environment. Therefore, it is important to determine which virtual machine software is capable of running Cage. In each test case the modifications to the 3.9.4 version of the Linux kernel were made and an identical version of Wireshark was used to capture the packets. A test program was created that used the modified `mmap` system call to create a caged vm_area. A loop was used that generated a set number (100) of memory events in this caged vm_area. Wireshark was set to capture from the memory network device and the test program was executed. The number of packets captured was noted as well

as the number of packets dropped, according to the statistics reported by Wireshark. (Tcpdump was also used to double check the statistics reported by Wireshark.) A visual inspection was also made of the memory event packets produced to ensure the correctness of the information contained in the memory event packets. The kernel error log was also checked to ensure that no errors had been reported. Figure 4.11 shows the results running on Intel hardware and Figure 4.12 shows the results running on AMD hardware.

Architecture	Virtual Machine	Host System	Guest System	Dropped/Missing Packets	Other Issues
Intel	N/A	Centos 6.5	N/A	No	No
	Virtual Box 4.2.16	Mac OSX	Centos 6.5	Yes	No
		Centos 6.5	Centos 6.5	Yes	No
	Virtual Box 4.3.8	Centos 6.5	Centos 6.5	N/A	Trace/Breakpoint Trap
	VmWare Fusion 5.0.0	Mac OSX	Centos 6.5	No	No
	Parallels	Max OSX	Centos 6.5	No	No
	Xen (full-virt and para-virt)	Centos 6.5	Centos 6.4	N/A	Trace/Breakpoint Trap
	QEMU	Centos 6.5	Centos 6.5	No	No

Figure 4.11: Cage running on Intel hardware.

Architecture	Virtual Machine	Host System	Guest System	Dropped/Missing Packets	Other Issues
AMD	N/A	Centos 6.5	N/A	No	No
	Virtual Box 4.2.16	Kubuntu 12.04	Centos 6.5	No	No
		Centos 6.5	Centos 6.5	No	No
	Virtual Box 4.3.8	Centos 6.5	Centos 6.5	No	No
	VmWare Workstation	Kubuntu 12.04	Centos 6.5	No	No
	Xen (full-virt and para-virt)	Centos 6.5	Centos 6.5	N/A	Trace/Breakpoint Trap
	QEMU	Centos 6.5	Centos 6.5	No	No

Figure 4.12: Cage running on AMD hardware.

The results show that Cage is stable while running natively on both Intel and AMD hardware as well as when running on some virtual machine software such as VmWare, Parallels and QEMU. Of interest is the result of Cage running on the Xen hypervisor. On both Intel and AMD hardware, Cage experiences an unexpected trace/breakpoint trap on the first caged memory event which halts the execution of the test program. This same behaviour was also seen with Virtual Box 4.3.8 running on Intel hardware. Virtual Box 4.2.16 running on Intel hardware does not produce the correct amount of memory event packets. These memory event packets are not reported as being dropped. Indeed, a closer inspection revealed that the memory event packets for these memory events were not even generated. While it is clear that Cage is stable while running natively on both Intel and AMD hardware as well as when running on some virtual machine software it is also clear that Cage is not stable while running on other virtual machine software such as Xen and Virtual Box. In spite of the issues with both Xen and Virtual Box we consider Cage to be stable based on its ability to run without issue natively on both Intel and AMD hardware.

4.4 Comparing Memory Event Packet Traces

Cage was tested to determine whether or not it is possible to compare two memory event packet traces from the same process to each other in order to validate the correctness of the packet trace. The goal of this test was to determine if it was possible to develop a "good" or "trusted" set of memory event packet traces for a set of processes in order to detect any future deviations from these trusted executions. In particular, this test was designed to be used as a test for the stability of Cage when run under different virtual machine software.

To determine whether or not it is possible to compare two memory event packet traces from the same process to each other, the same program, `whoami`, was executed

using the `cage_wrapper` to cage 100% of the process address space. The memory event packets from `whoami` were captured using Wireshark and the resulting file was saved. The `whoami` program was then executed again on the same machine and the memory event packets were again captured using Wireshark and saved. In each case, the number of packets captured was verified to be the same and no packets were reported as having been dropped by Wireshark. We created a program `diff_cage` that opened both files and compared the contents of each memory event packet in one file to the corresponding memory event packet in the other file. The items from the memory event packet that were compared were the following:

1. Instruction pointer
2. Effective address
3. Instruction pointer label
4. Effective address label
5. PTE meta-data
6. Read/write/execute flags
7. Instruction
8. Data contained at the effective address

The remaining elements of the memory event packet are known to either change, such as the PID, or not change, such as the UID, between executions of the same program.

The first approach to comparing the memory event packets to each other was to directly compare the values of the fields outlined above. This resulted in differences being noticed in the instruction pointer, the effective address, and in the data contained at the effective address. These results were expected because of address space layout randomization (ASLR). Indeed, when the test was performed with ASLR disabled there were no differences between the instruction pointers and the effective addresses in the memory event packet traces. The challenge then, was to compare

the memory event packet traces to each other while ASLR was enabled. The first attempt at managing ASLR was to take the offset for the instruction pointer or the effective address from the beginning of the vm_area the address was in. The vm_area creation packets were used to determine the start and end address of the vm_areas. The offsets were then compared to each other and the instruction pointer or effective address was considered the same if the offsets matched. This resulted in some, but not all of the instruction pointers and effective addresses matching between memory event packet traces.

The next approach was to store the first address accessed in a vm_area and take the offset between the first address and the current address. These offsets were again compared and the instruction pointer or the effective address was considered the same if the offsets matched between memory event packet traces. The result was that all of the instruction pointer addresses now matched between memory event packet traces and more of the effective addresses matched but not all. The remaining effective addresses that did not match were addresses on the stack. The final approach was to apply a combination of both techniques. In this approach all instruction pointer addresses and effective addresses were matched successfully. The combination of both taking the offset from the beginning of the vm_area and taking the offset from the first address accessed in the vm_area was sufficient to manage ASLR when comparing memory event packet traces.

The data contained at the effective address that differed was mainly composed of addresses as well as large values. In comparing the large values we noticed that the low byte of these values always matched. The differences between memory event packet traces seen in this case is caused by Cage arbitrarily reading eight bytes of the data contained at the effective address regardless of the size of the data actually contained at the effective address. If the size of the data contained at the effective

address is only one byte in length then Cage will still read eight bytes of data and will therefore have one byte of actual data and seven bytes of unknown data. The unknown data could be the contents of another variable or it could be unallocated memory.

The remaining memory event packets that do not compare with regards to the data contained at the effective address were composed of addresses. An attempt was made to compare these values by once again taking the offset from the first time an address in a vm_area was in the data contained at the effective address to the current data contained at the effective address and comparing the offsets. The result was that some of the addresses in the data contained at the effective address could be compared in this fashion. The offset between the addresses in the data contained at the effective address and the start of the vm_area containing that address were also compared in combination with the first result. The result of comparing both the offset from the current address seen in the data at the effective address with the first address seen and with the start of the vm_area containing the current address was that most of the differences between the data at the effective address in the memory event packet traces were accounted for. The remaining addresses that could not be accounted for contained in the data at the effective address were high memory addresses which did not belong to a caged vm_area.

The results of attempting to compare two memory event packet traces show that it is inherently difficult because of ASLR. In particular the difficulties in comparing the data contained at the effective address show that while it is possible to account for the differences seen, it is not possible to determine for certain that two memory event packet traces are identical.

4.4.1 Differences in Memory Event Packet Traces Between Machines

The problem of comparing memory event packet traces is made even more difficult when the process is executed on different machines. We performed another test in which we created a C program that simply returned from the main function on two identical native machines. The program was compiled separately on each machine and was dynamically linked on both machines. Both machines had ASLR enabled and prelinking enabled. When the memory event packets were captured on each machine, the number of memory event packets produced differed for the same program between the machines. The statistics reported by Wireshark were checked to ensure that neither machine was dropping packets. (These statistics were backed up by results from tcpdump.) An analysis of the packets showed that during the glibc startup code, different branches were being taken on one machine than on the other. These branches all corresponded to features of the processor, such as SSE, being present. A closer examination of our native machines revealed that while the machines were identical, the processors had slightly different revision numbers which accounted for some features of the processor being available on one machine but not the other machine. This created a different number of memory event packets being produced for the same program across different machines.

4.5 Summary

This chapter discussed the evaluations that were performed on the Cage mechanism. The first section discussed the results of five different performance tests that were performed. These tests highlighted the performance of various use cases of the Cage mechanism as well as demonstrating the overhead of different components of the mechanism. The second section detailed our attempts at capturing 100% of the memory event packets and the relative magnitude of information contained within a

memory event packet trace. The third section discussed the generality of the Cage mechanism to various platforms including different virtual machines and running natively on machines. The fourth section discussed our attempts at comparing two memory event packet traces and the difficulties we had due to ASLR. The next chapter will discuss the applications of the Cage mechanism including the different BPF filters that we created.

Chapter 5

Applications of Cage

As an application of the Cage mechanism, we have developed eleven different BPF filters that can be used to detect varying types of patterns in the memory event stream. Section 5.1 describes these eleven BPF filters in detail. We have also developed a program that can automatically find specific patterns in the memory event stream and output relevant information about these patterns as discussed in Section 5.2. In Section 5.3 we discuss how the BPF filters and our pattern finding program can be combined to enforce patterns in the memory event stream at runtime. Finally Section 5.4 discusses some limitations to finding patterns with the Cage mechanism as well as possible solutions for these limitations.

5.1 BPF Filters

This section details the varying BPF filters that we have created. It is important to understand the types of filters that have been created as well as their strengths and limitations. The remaining sections of this chapter refer to these filters. All of these filters are designed to look for specific patterns in the memory event stream. While these filters may seem simplistic, i.e. we aren't looking for a buffer overflow or other such memory errors, we view the filters we have created as building blocks for larger, more complex filters. We believe it is important to firmly establish the reliability of these building block filters before attempting to compose them into a larger filter.

Temporal Filtering

The purpose of this filter is to induce a type of rate limiting on the number of memory event packets that are generated. There are three different types of this filter. One

```
L1:  A = current effective address
L2: (A >= begin_address) ? goto L3 : goto L10
L3: (A > end_address) ? goto L10 : goto L4
L4:  A = A - end_address
L5: (A > end_address) ? goto L7: goto L6
L6: (A == end_address) ? goto L7: goto L10
L7:  A = current number of events
L8: (A == num) ? goto L9 : goto L11
L9:  return 1
L10: return 0
L11: return -1
```

Figure 5.1: Temporal filter. This filter will emit a memory event packet for every n^{th} packet as specified by num provided that memory event falls into a specific range of addresses specified by begin_address and end_address.

that watches for only read accesses, one that watches for only write accesses, and one that watches for either read or write accesses. This can be used to only produce a packet every n^{th} event. For example, if we know that the n^{th} write to an array is important we can ignore the first n-1 events. This filter watches for memory events that fall into a specified range of addresses. If the memory event falls within this range, the filter loads the current number of memory events that have been seen that fall within that range. If this stored value is equal to the number of events we are waiting for, then a memory event packet is created. Since BPF has no facility for a "jump less-than" we are forced to check the address range by checking to see if the current effective address minus the end_address is greater than the end_address. If it is greater than the end_address then the current effective address is within the range we are looking for. If it is not greater than the end_address than the current effective address is outside the range we are looking for. This filter is supported by code in Cage that maintains a piece of state storing how many events of this type have already occurred. Each time the specified number of events is reached, the stored number of events is set to zero by supporting code in Cage. Figure 5.1 shows the logic of this BPF filter.

```
L1:  A = current effective address
L2: (A == address) ? goto L3 : goto L4
L3:  return num
L4:  return 0
```

Figure 5.2: Data-overwriting filter. This filter will overwrite the data stored at the effective address specified by address with the value num.

Overwriting Data at an Effective Address

The purpose of this filter is to overwrite the data contained at a specific effective address with a different specified value. This can be used to change the contents of the memory of a program at runtime. This filter watches for a memory event corresponding to a specific effective address. When this effective address is accessed during program execution, the filter returns the value that will be used to overwrite the data contained at the effective address. This is supported by code in Cage that watches for a non-zero return from the filter. (It is assumed that the value used for overwriting the data contained at the effective address will not be zero.) Cage then takes the value returned by the filter and overwrites the data contained at the current effective address, using a Cage-specific "copy_to_user" function, with this value. Figure 5.2 shows an example of the logic of this BPF filter.

Overwriting an Instruction

The purpose of this filter is to overwrite an instruction at a specific instruction pointer with a different specified instruction. This can be used to modify the executing instruction of a program at runtime. The new instruction must be the same size as the old instruction to prevent overwriting subsequent instructions. Overwriting an instruction in this way has the side effect of modifying the binary so that each subsequent run of the binary executes the new instruction. This filter watches for a memory event corresponding to a fetch of a particular instruction specified by the given instruction pointer. When this memory event occurs, the filter returns the

number of bytes contained in the new instruction. This is the number of bytes that will be written to the location of the current instruction pointer. The new instruction is passed to Cage through a filter structure that contains the BPF instructions along with the type of BPF filter we are executing. This is supported with code in Cage that recognizes the non-zero return value from the filter and overwrites the instruction contained at the current instruction pointer with the new instruction. This overwriting occurs during the memory event corresponding to the fetch of the instruction. This BPF filter is identical to Figure 5.2 except that we load the instruction pointer in the first step instead of the effective address.

Viewing a Buffer Allocated at Runtime

The purpose of this filter is to view all the memory events corresponding to a specific buffer that has been allocated at runtime. In this instance we cannot know the effective address to search for ahead of time. This filter gives us the ability to locate a dynamically allocated data structure of interest and recover all the memory events from the point of its creation on. This filter relies on the input of a specific instruction pointer. The value contained in the register rax at this instruction must be the address of a buffer. This filter watches for the memory event corresponding to the execution of a specific instruction pointer. When this memory event occurs, the value contained in the rax register at this time is loaded into the BPF filter and this value is returned. Supporting code in Cage then recognizes this return value and stores the returned value in a piece of state. The rest of the filter is executed any time the current instruction pointer does not match the specified instruction pointer. This part of the filter loads the stored rax value and checks to see if the current effective address is within the range of the stored rax value plus a specified value which is the length of the buffer to search for. If it is within this range then the memory event corresponds to the buffer we are looking at and a packet is emitted. We have used this filter to

```
L1:   A = current instruction pointer
L2:  (A == address) ? goto L3 : goto L5
L3:   A = current value of rax
L4:   return A
L5:   A = stored value of rax
L6:   X = A
L7:   A = current effective address
L8:  (A >= X) ? goto L9 : goto L15
L9:   A = X
L10:  A = A + num
L11:  X = A
L12:  A = current effective address
L13: (A > X) ? goto L15 : goto L14
L14:  return 1
L15:  return 0
```

Figure 5.3: Viewing memory events for a buffer allocated at runtime. This filter will produce memory event packets for every memory event corresponding to a dynamically allocated buffer.

recover the ssh private key from the ssh client while it is connecting to an ssh server. Figure 5.3 shows an example of the logic of this BPF filter.

Capturing the Buffers in a Program

The purpose of this filter is to produce memory event packets corresponding to every buffer in a program that gets accessed sequentially. There are two types of this filter, one that looks for successive reads from a buffer and one that looks for successive writes to a buffer. This filter compares the stored effective address with the current effective address and produces a packet if the two addresses are eight bytes away from each other. Eight bytes was chosen under the assumption that most sequential reads and writes to a buffer are optimized to read or write eight bytes at a time on a 64-bit system. This filter is supported by code in Cage that updates the value of the stored effective address each time the filter completes. Figure 5.4 shows an example of the logic of this BPF filter.

```
L1: A = stored value of effective address
L2: X = A
L3: A = current value of effective address
L4: A = A - X
L5: (A == 0x8) ? goto L6 : goto L7
L6: return 1
L7: return 0
```

Figure 5.4: Filter to find buffers in a program. This filter will search for sequential read or write accesses to buffers within a program and emit memory event packets for these buffers.

Write, Followed by Reads

The purpose of this filter is to produce memory event packets for an effective address that is accessed in the pattern of a write followed by some number of reads. It is assumed that the effective address is either known by the filter or that each new effective address encountered is the effective address to look at for the pattern. Unlike the previous filters, there is no code in Cage that supports this filter. This filter is able to run using only the extensions that were made to BPF. We created a test program that produced exactly the pattern of one write to an effective address followed by some number of reads. This filter correctly produced memory event packets for only the pattern specified. Figure 5.5 shows an example of the logic of this BPF filter and represents a finite state machine.

Range-Checking Filter

The range-checking filter will ensure that the data contained at the effective address is within a certain range. As an example, the filter in Figure 5.6 is checking to ensure that the data contained at the effective address is within the range of 0–10. The effective address is not checked in this example; checking the effective address has been demonstrated in previous filters and can be added on if necessary. We created a test program that produced memory events that modified the data contained at

```
L1:   A = R/W/X flags
L2:  (A == Write) ? goto L3 : goto L9
L3:   A = stored value 1
L4:   X = A
L5:   A = current value of effective address
L6:  (A == X) ? goto L17 : goto L7
L7:   Stored value 1 = current effective address
L8:   return 1
L9:   A = stored value 1
L10:  X = A
L11:  A = current value of effective address
L12: (A == X) ? goto L13: goto L16
L13:  A = R/W/X Flags
L14: (A == Read) ? goto L15: goto L17
L15:  return 1
L16:  return 0
L17:  return -1
```

Figure 5.5: Filter of the pattern write, followed by reads. This filter produces memory event packets for the pattern of one write followed by any number of reads.

the effective address within the specified range. The range-checking filter correctly produced the memory event packets for only the pattern specified. In constructing this filter and testing it we noticed that when attempting to monitor the data contained at the effective address, write events showed the *previous* value for the data contained at the effective address and not the value that was being written. This is due to the location in the program control flow where memory event packets are created. Specifically, as discussed previously, memory event packets are created before the instruction that is executing is allowed to complete. This means that the write to the data at the effective address has not occurred when the memory event packet is created. This results in an "off by one" error in the range-checking filter. If the last memory event is a write event that writes a value larger than 10 into the data contained at the effective address the range-checking filter will not be able to detect this. At the beginning of the filter logic, A is loaded with the value 0xFFFFFFFF and then left-shifted by 32. This is necessary to create a 64-bit value in A because the

```
L1:   A = 0xFFFFFFFF
L2:   A = A left-shifted by 32
L3:   A = A | 0xFFFFFFF6
L4:   X = A
L5:   A = data contained at effective address
L6:  (A == 0) ? goto L10: goto L7
L7:   A = A - 10
L8:  (A == 0) ? goto L10: goto L9
L9:  (A >= X) ? goto L10: goto L11
L10: return 1
L11: return -1
```

Figure 5.6: Range-Checking filter. This filter produces memory event packets while the data contained at the effective address is within the range 0–10.

```
L1:   A = R/W/X flags
L2:  (A == write) ? goto L3: goto L4
L3:  return 1
L4:  return -1
```

Figure 5.7: Always Read/Written Filter. This filter produces memory event packets if the effective address is either always read from or written to. In this example the effective address is always written to.

value of K is only 32 bits. The value of 0xFFFFFFFF is the value of K. Figure 5.6 shows an example of the logic of the range-checking filter.

Always Read/Written Filter

The always read or always written filter produces memory event packets if the effective address is either always read from or always written to. It is assumed that we are told whether we are watching for always reads or always writes. Once again the effective address is not checked and it is assumed that the first effective address is the one we are watching for the pattern of always read from or always written to. Figure 5.7 shows an example of the logic of the filter when watching for writes.

```
L1:  A = stored value 2
L2: (A == 0) ? goto L5: goto L3
L3: (A == 1) ? goto L11: goto L4
L4: (A == 2) ? goto L5: goto L10
L5:  A = R/W/X flags
L6: (A == write) ? goto L7: goto L10
L7:  A == 1
L8:  Stored value 2 = A
L9:  return 1
L10: return -1
L11: A = R/W/X flags
L12: (A == read) ? goto L13: goto L16
L13: A = 2
L14: Stored value 2 = A
L15: return 1
L16: return -1
```

Figure 5.8: Write, Read, Repeat. This filter produces memory event packets if the effective address is accessed in alternating write and read events beginning with a write.

Write, Read, Repeat Filter

The write, read, and repeat filter produces memory event packets if the effective address is accessed in an alternating pattern of writes and reads beginning with either a write or a read. This filter makes use of the stored memory locations that were created as an extension of BPF to create a flag indicating the type of the previous memory event. Again, the effective address is not checked and it is assumed that the first effective address is the one we are watching for the pattern of writes and reads. This filter requires that the stored memory locations are cleared after the end of each program execution to prevent the previous program interfering with the next program to be executed. Figure 5.8 shows an example of the logic of this filter where the pattern begins with a write.

```
L1:  A = instruction pointer
L2: (A == 0x400513) ? goto L7: goto L3
L3: (A == 0x40051e) ? goto L7: goto L4
L4: (A == 0x400537) ? goto L7: goto L5
L5: (A == 0x40053e) ? goto L7: goto L6
L6: (A == 0x400552) ? goto L7: goto L8
L7: return 1
L8: return -1
```

Figure 5.9: Short list of instructions. This filter produces memory event packets if the instruction pointer is in the list of instructions that accesses a given effective address.

Short List of Instructions Filter

The short list of instructions filter watches for specific instruction pointers accessing the same effective address. It is assumed that the instruction pointer values are known and so may be hard-coded into the filter. This is similar to AccMon's [53] "Program Counter-Based Invariants", which are the set of instructions that access a particular effective address, referred to as an "AccSet". This filter watches for instruction pointers that are in the effective addresses AccSet. The instruction pointers hard-coded into the example are taken from the test program that we wrote to create this pattern. Figure 5.9 shows an example of the logic of this BPF filter.

Monotonically Increasing/Decreasing Filter

The monotonically increasing or decreasing filter watches the data contained at a specific effective address to determine if it is monotonically increasing or decreasing. As with the range-checking filter, the monotonically increasing or decreasing filter also has the problem of a write memory event not showing the value being written. This is especially problematic if the first memory event is a write event. Before the first write memory event occurs, the data contained at the effective address appears to be zero. It is not until after the first write memory event that this value is updated. Not being able to determine the first value that was written to the data contained at the effective

```
L1:   A = stored value 1
L2:  (A == 0) ? goto L3: goto L6
L3:   A = data contained at the effective address
L4:   Stored value 1 = A
L5:   return 1
L6:   X = A
L7:   A = data contained at the effective address
L8:  (A > X) ? goto L10: goto L9
L9:  (A == X) ? goto L11, goto L12
L10: Stored value 1 = A
L11: return 1
L12: return -1
```

Figure 5.10: Monotonically increasing/decreasing. This filter produces memory event packets if the data contained at the effective address is monotonically increasing.

address prevents us from noticing what the data contained at the effective address was initialized by the first write memory event. This interferes with determining if the value contained in the data at the effective address is monotonically increasing or decreasing as it always appears to start at zero. Further adding to the complexity of this filter is the inability to determine when an overflow or underflow has occurred in the data contained at the effective address. Without knowing that an overflow or underflow has occurred it appears to the BPF filter that the value contained in the data at the effective address is suddenly zero again which will fail the check for monotonicity. Figure 5.10 shows the logic for the monotonically increasing BPF filter.

5.2 Pattern Finding Program

In addition to creating the above BPF filters to recognize patterns in the memory event stream we also created a program that will find the following patterns in the memory event stream.

1. Write, followed by some number of reads
2. Range of the data contained at the effective address

3. Always read from or written to
4. Alternating write and read events
5. Short list of instructions
6. Monotonicity

This pattern finding program `find_patterns` does offline analysis on the memory event packets that have been captured and saved from a previous run of a program. Offline analysis was chosen because detecting patterns while capturing the memory event packets of a currently executing program would inhibit the capture speed, possibly to the point that our capture utility would not be able to capture 100% of the memory event packets.

For each effective address in the memory event stream, the previous list of patterns are evaluated to determine if that pattern resides at the current effective address. Any memory event that does not match the pattern being evaluated moves that pattern into a fail state for that effective address indicating that the pattern does not reside at that effective address. The program `find_patterns` also prints out the patterns contained at each effective address as well as relevant information about the pattern such as the instruction pointers that accessed the effective address or the range of values contained in the data at the effective address.

For each type of pattern listed above, we created a test program that produces exactly the pattern of memory event packets listed. These test programs were used to aid in constructing the BPF filters discussed previously as well as the pattern matching code of `find_patterns`. In addition, both `find_patterns` and the BPF filters were tested on the memory event packets captured during the execution of these test programs to ensure that the patterns could be recognized.

As a larger test, `find_patterns` was also run over the memory event packets saved from an execution of the program `whoami`. This test demonstrated that multiple

patterns can reside at the same effective address. For example, it is possible for an effective address to be always written to and be monotonically increasing or decreasing at the same time. This makes it difficult or impossible to narrow down the patterns contained at an effective address to one pattern which indicates that either multiple BPF filters must be used or a single larger filter that can watch for all of the patterns at the effective address. In addition, some effective addresses are only accessed once which does not match any of the above patterns. Also, each effective address has a short list of instructions that access it. In `whoami` this list of instructions was relatively short (2–5 instructions) for each effective address, reinforcing the idea of AccMon's [53] "Program Counter-Based Invariants" in that it is possible to create a short list of instructions that access each effective address.

As discussed in Section 5.1 some filters or patterns are hard to match exactly because of the limitations of the memory event packets. The program `find_patterns` has the same limitations regarding overflow and write events. Overflow or underflow cannot be detected which means that when either of these occur it appears to `find_patterns` as if the value suddenly went to zero instead of a larger or smaller number. This creates a problem in detecting the range of values contained at the effective address as well as in detecting monotonicity. Write events also affect these patterns as the memory event packet does not contain a field indicating what value is about to be written. This can create an "off by one" error in detecting these patterns as `find_patterns` has to wait for the next memory event at that effective address to notice what was written. If the write event writes a value that breaks the pattern it will still be noticed, provided that it was not the last memory event packet for that effective address, but it will break the pattern one memory event packet too late. The inability to see what value is being written also affects determining what an effective address is being initialized to by the first write event. The data at the effective ad-

dress appears to be zero until it is initialized. This can also confuse the range-checking pattern and the monotonically increasing or decreasing pattern as it appears that the data at the effective address always begins at zero. Nevertheless `find_patterns` can successfully identify the varying patterns at the effective addresses and output information relevant to each pattern. This information can then be used with the BPF filters to fine tune them for a specific effective address.

5.3 Pattern Finding/Enforcing BPF Filters

The BPF filters we created and `find_patterns` gives us the ability to find patterns in the memory event stream. However, the BPF filters can also give us the ability to enforce the patterns during the execution of a process. The position at which the BPF filter is evaluated, before the instruction executes, gives us the ability to prevent instructions from executing that do not match the pattern. Knowing that a specific pattern exists at an effective address allows us to use the BPF filters to watch for events that do not conform to that pattern. In the example BPF code shown in previous figures, the BPF filters return -1 when a memory event does not match the pattern. This return value will cause a segmentation fault to be raised and the executing program will be terminated.

We used the test programs that were created for each BPF filter to produce memory events corresponding to a specific pattern to test the enforcement capability of the BPF filters. We modified the test programs slightly so that they produced at least one memory event that did not match the pattern the BPF filter was watching for. In each case the BPF filter correctly identified the out of place memory event and returned a -1 causing a segmentation fault to be raised. The combination of the BPF filters and `find_patterns` produces a work flow wherein `find_patterns` can be used to find the patterns contained at an effective address and output relevant information

about the pattern such as its range. This information can be used to inform the BPF filters of the correct parameters for the pattern. The BPF filters can then be used to enforce these patterns during the execution of the process.

5.4 Limitations of Finding Patterns/BPF Filters

In the sections above, we encountered and mentioned some limitations that occur with some of the patterns. In this section we will discuss these limitations as well as possible solutions for these limitations.

Overflow and Underflow

As mentioned previously, Cage has no facility to detect either overflow or underflow in the data contained at the effective address. This plays a significant role in the ability of `find_patterns` and the BPF filters to determine things like the true range of values contained in the data at the effective address as well as monotonically increasing or decreasing values. A solution to this limitation would be the addition of a new type of memory event packet. This memory event packet would be generated from the debug fault handler. At this point the execution of the current instruction has already occurred, and the flags in the EFLAGS register will indicate the occurrence of an overflow or an underflow. Checking for the overflow or underflow condition and creating a memory event packet that indicates if an overflow or underflow occurred will allow both `find_patterns` and the BPF filters to recognize when an overflow or underflow occurs.

Addresses Changing Over Runs of a Program

Up until this point we have not mentioned the issues of the effective address changing between runs of a program due to ASLR. Clearly this poses a significant limitation with regards to enforcing the patterns that `find_patterns` has detected in that the

effective addresses reported by `find_patterns` will not be the same when the BPF filters are used to enforce these patterns. While we have not attempted to account for differing effective addresses in our BPF filters, we believe our work on comparing two memory event packet traces (Section 4.4, page 56) can inform us of how this could be done. While attempting to compare two memory event packet traces we determined that the effective addresses could be compared by using two methods. The first is to compare the current effective address with the first effective address seen to access the same vm_area. The second is to compare the current effective address with the start of the vm_area. When comparing memory event packet traces we determined that the combination of these two methods allowed us to compare the effective addresses across different runs of the same program. We believe that a BPF filter could be created that can do this comparison thus allowing us to use the BPF filters to enforce the patterns that are found by `find_patterns`.

"Off by One" Errors

In some filters, such as the range checking filter, an "off by one" error can occur because the last value that is written to the data contained at the effective address is not shown. This is because the memory event packets contain the current value of the data contained at the effective address; they do not show what value is being written to the data at the effective address. This limitation can also be overcome by adding in a second memory event packet on a write memory event that is generated in the debug fault handler. At this point the execution of the current instruction has already taken place and the data contained at the effective address will already be modified by the write memory event. This second write memory event packet will contain the new value of the data contained at the effective address. This would allow `find_patterns` and the BPF filters to accurately detect what value is being written to the data contained at the effective address which would eliminate the "off by one"

error.

5.5 Summary

This chapter discussed the eleven BPF filters that we created in detail in Section 5.1. Section 5.2 discussed our program to find patterns in the memory event stream while Section 5.3 discussed how to combine the pattern finding program and the BPF filters to enforce patterns in the memory event stream. Finally, Section 5.4 discussed some limitations to finding patterns in the memory event stream as well as some possible solutions for these limitations. The next chapter will conclude.

Chapter 6

Conclusion and Future Work

In this thesis we presented Cage, a kernel-level mechanism for both intercepting and filtering the memory events of a user-level process. As described in Chapter 3, Cage exists as a series of modifications to the Linux kernel. These modifications cover 23 different files and over 1000 lines of additional code in the Linux kernel in addition to supporting loadable kernel modules and user-level programs. In the next section we discuss some future work and finally, Section 6.2 concludes.

6.1 Future Work

There are many areas where we would like to expand or improve Cage. Currently Cage has not been tested with more than one program caged at a time. We would like to cage multiple programs to ensure that the Cage mechanism still functions as expected under these conditions. We also have not tested a program that uses multiple threads during execution. While we do not foresee any difficulties with these two test cases, we would still like to test them to ensure our expectations are met.

At this point in time Cage has the ability to label the pages of a caged region of memory. This gives us a label for the page containing the effective address as well as a label for the page containing the instruction pointer. Currently, these labels are not used other than to demonstrate that it is possible to label a page using unused bits in the PTE. We would like to expand the mechanisms that currently label these pages to be more usable and programmable. We would also like to use these labels in some fashion, for example, as a way to enforce a security policy on memory.

Cage currently employs a global BPF filter to filter the memory event stream of a

process. This is a limiting design because it means that only one filter can be in place at a time and that filter will apply to all caged processes. We would like to employ filters on a per-process basis. That is, there would be one filter per process. This would allow multiple caged programs to execute concurrently using different filters.

As mentioned in Section 4.1, the mcf benchmark executes faster on the Cage kernel then it does on the standard kernel. We would like to investigate this case further and attempt to determine a cause for this behaviour.

Finally, we would like to perform an analysis to determine exactly which memory vulnerabilities Cage is able to detect.

6.2 Conclusion

In Chapter 1 we described four properties that define a principled approach to memory interception and filtering. The first property was the speed of the memory interception function and the filtering function. Ideally there would be no detectable slow-down of the target process. However, this ideal case would require a hardware solution to memory interception and filtering that, to our knowledge, currently does not exist. The second property was the transparency of the memory interception function and the filtering function. Again, in the ideal case the memory interception and the filtering function would be undetectable by the target process and so would not affect the behaviour of the target process. A more lightly transparent approach would require no modifications to the source code or binary image of the target process with the goal of not modifying the behaviour of the target process. The third property was the reliability of the memory interception function and the filtering function. Given the same program with the same inputs and executed on the same hardware, the memory interception function and the filtering function should produce the same record of execution across multiple runs of the program. The fourth property was

completeness and refers to the vantage point of the memory interception function and filtering function on the memory event stream. To be considered complete, the memory interception function and the filtering function must work on each instruction level memory event within the memory event stream. We maintain that Cage meets the principles of being lightly transparent, reliable, and complete.

In Chapter 4 we evaluated the performance of Cage and saw a significant slow-down in the runtime of the caged process. We expected this slow-down because of the overhead of handling a page fault for every memory access. In addition, the TLB flush that is required each time a page is re-caged effectively disables the TLB when a process is fully caged. Also, instruction prefetches are unable to handle a page fault and so instruction prefetching is effectively disabled as well if the .text section is caged. For this reason we do not claim that Cage meets the principle of speed, nor did we expect it to.

Cage does meet the principle of being lightly transparent, however. Cage exists as a series of modifications to the 3.9.4 version of the Linux kernel. In Chapter 3 we described the memory interception mechanism and the filtering mechanism that Cage uses. The memory interception mechanism causes a page fault to be generated for each memory event within a caged region of memory. The filtering mechanism uses a BPF filter to filter each memory event within the page fault handler. Neither the filtering mechanism nor the memory interception mechanism require any modifications to the caged program's source code or binary image. Also described in Chapter 3 are the supporting mechanisms that we have developed to create a caged process. With the exception of the instrumented `mmap` system call, none of the mechanisms to create a caged process require the modification of the target program's source code or binary image. We do not consider the instrumented `mmap` system call to be in violation of the principle of being lightly transparent because Cage does not require

its use to function. It is an option to developers should they wish to create a special caged region of memory.

Cage meets the principle of reliability. In Chapter 4, Section 4.4 we compared two memory event packet traces that were generated from the same program with the same inputs and on the same hardware. While there were challenges in comparing the memory event packet traces due to ASLR, it was also clear that the two memory event packet traces had the same memory event packets in the same order. If this was not the case then there would have been more differences between the memory event packet traces that we could not account for with ASLR. From this analysis it is clear that Cage is reliable.

Cage meets the principle of completeness. The mechanism of generating a page fault on every memory access including instruction fetches and read or write accesses to data ensures that every memory event is intercepted. The filtering mechanism exists within the page fault handler and is guaranteed to filter each memory event that is intercepted. Since the page fault memory interception mechanism guarantees that every instruction level memory event is intercepted, Cage is guaranteed to meet the principle of completeness in that Cage intercepts and filters every memory event from the memory event stream.

While Cage meets the principles of being lightly transparent, reliable and complete, ideally there would be a memory interception and filtering mechanism that meets the principles of speed, full transparency, reliability and completeness. This would require hardware primitives built into the memory management unit that provided the ability to intercept each memory event and filter each memory event. Until this support is added to commodity hardware, memory interception mechanisms and filtering mechanisms of the memory event stream will remain less than ideal.

Cage's ability to specify arbitrary BPF filters gives it a wide range of capabilities.

In Chapter 5 we showed examples of eleven different types of BPF filters that we have created. This is by no means the extent of the capabilities of the BPF filters. In general, the only limitation on the BPF filters is that of the Cage-extended form of BPF itself. This provides the ability to not only detect a wide range of patterns in the memory event stream but to enforce those patterns as well. This, combined with Cage's ability to run on commodity hardware and operating systems without requiring modifications to a target program's source code or binary image makes Cage a valuable security tool.

Bibliography

[1] Aamer Jaleel. Memory Characterization of Workloads Using Instrumentation-Driven Simulation. `http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf`. Date Accessed, June 18, 2015.

[2] Julian Bangert, Sergey Bratus, Rebecca Shapiro, Michael E. Locasto, Jason Reeves, Sean W. Smith, and Anna Shubina. ELFbac: Using the Loader Format for Intent-Level Semantics and Fine-Grained Protection. In *Dartmouth College Computer Science Technical Report TR2013-727*, June 2013.

[3] Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W. Smith. The Page-Fault Weird Machine: Lessons in Instruction-less Computation. In *the 7th USENIX Workshop on Offensive Technologies*, Berkeley, CA, 2013. USENIX.

[4] Bochs. `http://bochs.sourceforge.net`. Date Accessed June 18, 2015.

[5] Sergey Bratus, Michael E. Locasto, and Brian Schulte. SegSlice: Towards a New Class of Secure Programming Primitives for Trustworthy Platforms. In *International Conference on Trust and Trustworthy Computing (TRUST 10)*, pages 228–245, 2010.

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

[7] Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, November 2000.

[8] Michael Burrows, Stephen N. Freund, and Janet L. Wiener. Run-time type checking for binary programs. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 90–105, Berlin, Heidelberg, 2003. Springer-Verlag.

[9] The clone Team. clone - create a child process. Centos 6.4 man page, section 2, July 2009.

[10] Standard Performance Evaluation Corporation. SPEC CPU2006. `https://www.spec.org/cpu2006/`, 1995 - 2015.

[11] The GDB developers. Debugging with GDB. Project website. `http://sourceware.org/gdb/current/onlinedocs/gdb/`. Date Accessed June 18, 2015.

[12] The GDB developers. GDB: The GNU Project Debugger. Project website. `http://www.gnu.org/software/gdb/`. Date Accessed June 18, 2015.

[13] Bryan Ford and Russ Cox. Vx32: Lightweight User-level Sandboxing on the x86. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 293–306, Berkeley, CA, USA, 2008. USENIX Association.

[14] The Hex-Rays group. Hex-Rays IDA. Project website. `https://www.hex-rays.com/products/ida/`. Date Accessed June 18, 2015.

[15] halfdead. Mistifying the debugger, ultimate stealthiness. In *Phrack 65:8*.

[16] IBM. Rational Purify for Windows. Project website. `http://www-03.ibm.com/software/products/en/ratpurwin`. Date Accessed June 18, 2015.

[17] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 325462-046US. March 2013.

[18] Mateusz Jurczyk and Gynvael Coldwind. Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns. White paper, presentation at SyScan 2013, `http://vexillium.org/dl.php?bochspwn.pdf`, 2013.

[19] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging Operating Systems with Time-traveling Virtual Machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association.

[20] Sarah Laing, Michael E. Locasto, and John Aycock. An Experience Report on Extracting and Viewing Memory Events via Wireshark. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)*, San Diego, CA, August 2014. USENIX Association.

[21] Michael A. Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. PEBIL: Efficient Static Binary Instrumentation for Linux. In *IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, 2010.

[22] The ld.so Team. ld.so - dynamic linker/loader. Centos 6.4 man page, section 8, January 2009.

[23] Andreas Lobel. 429.mcf SPEC CPU2006 Benchmark Description. `https://www.spec.org/auto/cpu2006/Docs/429.mcf.html`. Date Accessed: June 19, 2015.

[24] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 190–200.

[25] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the USENIX Winter 1993 Conference*, USENIX'93, pages 2–2, Berkeley, CA, USA, 1993. USENIX Association.

[26] Mingwei Zhang and Rui Qiao and Niranjan Hasabnis and R. Sekar. A Platform for Secure Static Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 129–140, New York, NY, USA, 2014. ACM.

[27] The mmap Team. mmap - map or unmap files or devices into memory. Centos 6.4 man page, section 2, September 2009.

[28] Jeff Muizelaar and Pekka Paalanen. In-kernel memory-mapped I/O tracing. `https://www.kernel.org/doc/Documentation/trace/mmiotrace.txt`, 2007.

[29] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[30] Vegard Nossum. kmemcheck. `https://www.kernel.org/doc/Documentation/kmemcheck.txt`. Date Accessed, June 18, 2015.

[31] Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In 13^{th} *USENIX Security Symposium*, pages 179–194, 2004.

[32] The ptrace Team. Ptrace - process trace. Centos 6.4 man page, section 2, March 2009.

[33] Julian Seward and Nicholas Nethercote. Using Valgrind to Detect Undefined Value Errors With Bit-Precision. In *Proceedings of the USENIX 2005 Annual Technical Conference*, pages 17–30, April 2005.

[34] Shan Lu and Soyeon Park and Chongfeng Hu and Xiao Ma and Weihang Jiang and Zhenmin Li and Raluca A. Popa and Yuanyuan Zhou. MUVI: Automatically Inferring Multi-variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 103–116, New York, NY, USA, 2007. ACM.

[35] Stylianos Sidiroglou. *Software Self-healing Using Error Virtualization*. PhD thesis, Columbia University, 2008.

[36] Skape. Memalyze: Dynamic Analysis of Memory Access Behavior in Software. `http://uninformed.org/?v=7&a=1&t=sumry`, 2007. Date Accessed, June 18, 2015.

[37] Joe Stewart. OllyBone: Semi-Automatic Unpacking on IA-32. In *DEFCON 14*, 2006.

[38] The strace Team. Strace - trace system calls and signals. Centos 6.4 man page, section 1, January 2003.

[39] PaX Team. PAGEEXEC. `http://pax.grsecurity.net/docs/pageexec.old.txt`, 2003. Date Accessed, June 18 2015.

[40] The Bochs team. Bochs User Manual. Project website. `http://bochs.sourceforge.net/cgi-bin/topper.pl?name=New+Bochs+Documentation&url=http://bochs.sourceforge.net/doc/docbook`. Date Accessed: December 9, 2014.

[41] The DTrace Team. DTrace: illumos Dynamic Tracing Guide. http://dtrace.org/guide/bookinfo.html. Date Accessed, June 18 2015.

[42] The DynamoRIO Team. DynamoRIO Dynamic Instrumentation Tool Platform. http://www.dynamorio.org/home.html. Date Accessed, June 18 2015.

[43] The ERESI team. The ERESI Reverse Engineering Software Interface. Project website. http://www.eresi-project.org/wiki/. Date Accessed June 18, 2015.

[44] The Nouveau Team. Nouveau: Accelerated Open Source driver for nVidia cards. http://nouveau.freedesktop.org/wiki/. Date Accessed June 18, 2015.

[45] The QEMU team. QEMU: Open Source Processor Emulator. Project website. http://wiki.qemu.org/Main_Page. Date Accessed June 18, 2015.

[46] The Wireshark Team. Dumpcap - dump network traffic. Project man page https://www.wireshark.org/docs/man-pages/dumpcap.html. Date Accessed June 18, 2015.

[47] The Wireshark Team. Wireshark. Project Website https://www.wireshark.org/. Date Accessed June 18, 2015.

[48] Vivek Thampi. Udis86 - disassembler library for x86 and x86_64. https://github.com/vmt/udis86. Date Accessed, June 18, 2015.

[49] Steven McCanne Van Jacobsen, Craig Leres. Tcpdump - dump traffic on a network. Centos 6.4 man page, section 8, March 2009. Currently maintained by tcpdump.org.

[50] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging. In *IEEE 13th International Symposium on High Performance Computer Architecture. HPCA 2007*, pages 273–284, Feb 2007.

[51] Oleh Yuschuk. OllyDbg. `http://www.ollydbg.de/`, 2000 - 2014.

[52] Michal Zalewski. Fenris - Program Execution Path Analysis Tool. Project website. `lcamtuf.coredump.cx/fenris/README`. Date Accessed: June 16, 2015.

[53] Pin Zhou, Wei Liu, Long Fei, Shan Lu, Feng Qin, Yuanyuan Zhou, Samuel Midkiff, and Josep Torrellas. AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-Based Invariants. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 269–280, Washington, DC, USA, 2004. IEEE Computer Society.

[54] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 224–236, Washington, DC, USA, 2004. IEEE Computer Society.

Appendix A

Raw BPF Filter Code

Temporal Filter (cf. Figure 5.1)

```
BPF_STMT(BPF_MISC+BPF_LDC, BPF_GET_EA)
BPF_JUMP(BPF_JMP+BPF_JGE+BPF_K, begin_address,0,7)
BPF_JUMP(BPF_JMP+BPF_JGT+BPF_K, end_address,6,0)
BPF_STMT(BPF_ALU+BPF_SUB+BPF_K, end_address)
BPF_JUMP(BPF_JMP+BPF_JGT+BPF_K, end_address, 1, 0)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0, 0, 3)
BPF_STMT(BPF_MISC+BPF_LDC, BPF_GET_CURRENT_EVENTS)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, num, 0, 2)
BPF_STMT(BPF_RET+BPF_K, 1)
BPF_STMT(BPF_RET+BPF_K, 0)
BPF_STMT(BPF_RET+BPF_K, -1)
```

Data-Overwriting Filter (cf. Figure 5.2)

```
BPF_STMT(BPF_MISC+BPF_LDC, BPF_GET_EA)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, address, 1, 0)
BPF_STMT(BPF_RET+BPF_K, 0)
BPF_STMT(BPF_RET+BPF_K, num)
```

Instruction-Overwriting Filter (cf. Section 5.1)

```
BPF_STMT(BPF_MISC+BPF_LDC, BPF_GET_EIP)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, address, 1, 0)
BPF_STMT(BPF_RET+BPF_K, 0)
BPF_STMT(BPF_RET+BPF_K, num)
```

Buffer-Viewing Filter (cf. Figure 5.3)

```
BPF_STMT(BPF_MISC+BPF_LDC, BPF_GET_EIP)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, address, 0, 2)
```

```
BPF_STMT(BPF_MISC+BPF_LDC, BPF_GET_RAX)
BPF_STMT(BPF_RET+BPF_A, 0)
BPF_STMT(BPF_MISC+BPF_LDC, BPF_GET_STORED_EA)
BPF_STMT(BPF_MISC+BPF_TAX, 0)
BPF_STMT(BPF_MISC+BPF_LDC, BPF_GET_EA)
BPF_JUMP(BPF_JMP+BPF_JGE+BPF_X, 0, 0, 6)
BPF_STMT(BPF_MISC+BPF_TXA, 0)
BPF_STMT(BPF_ALU+BPF_ADD+BPF_K, num)
BPF_STMT(BPF_MISC+BPF_TAX, 0)
BPF_STMT(BPF_MISC+BPF_LDC, BPF_GET_EA)
BPF_JUMP(BPF_JMP+BPF_JGT+BPF_X, 0, 1, 0)
BPF_STMT(BPF_RET+BPF_K, 1)
BPF_STMT(BPF_RET+BPF_K, 0)
```

Buffer-Finding Filter (cf. Figure 5.4)

```
BPF_STMT(BPF_MISC+BPF_LDC, BPF_GET_STORED_EA)
BPF_STMT(BPF_MISC+BPF_TAX, 0)
BPF_STMT(BPF_MISC+BPF_LDC, BPF_GET_EA)
BPF_STMT(BPF_ALU+BPF_SUB+BPF_X, 0)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x8, 0, 1)
BPF_STMT(BPF_RET+BPF_K, 1)
BPF_STMT(BPF_RET+BPF_K, 0)
```

Write, Followed by Reads Filter (cf. Figure 5.5)

```
BPF_STMT(BPF_MISC+BPF_LDC,BPF_GET_RWX)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x02, 0, 6)
BPF_STMT(BPF_MISC+BPF_LDC, BPF_LOAD_STORED_1)
BPF_STMT(BPF_MISC+BPF_TAX,0)
BPF_STMT(BPF_MISC+BPF_LDC,BPF_GET_EA)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_X,0,10,0)
BPF_STMT(BPF_MISC+BPF_STC,1)
BPF_STMT(BPF_RET+BPF_K, 1)
BPF_STMT(BPF_MISC+BPF_LDC,BPF_LOAD_STORED_1)
```

```
BPF_STMT(BPF_MISC+BPF_TAX,0)
BPF_STMT(BPF_MISC+BPF_LDC,BPF_GET_EA)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_X,0,0,3)
BPF_STMT(BPF_MISC+BPF_LDC,BPF_GET_RWX)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x00, 0, 2)
BPF_STMT(BPF_RET+BPF_K, 1)
BPF_STMT(BPF_RET+BPF_K, 0)
BPF_STMT(BPF_RET+BPF_K, -1)
```

Range-Checking Filter (cf. Figure 5.6)

```
BPF_STMT(BPF_LD+BPF_IMM,0XFFFFFFFF)
BPF_STMT(BPF_ALU+BPF_LSH+BPF_K, 32)
BPF_STMT(BPF_ALU+BPF_OR+BPF_K, 0XFFFFFFF6)
BPF_STMT(BPF_MISC+BPF_TAX,0)
BPF_STMT(BPF_MISC+BPF_LDC,BPF_GET_DATA_EA)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0, 3, 0)
BPF_STMT(BPF_ALU+BPF_SUB+BPF_K, 10)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0, 1, 0)
BPF_JUMP(BPF_JMP+BPF_JGE+BPF_X, 0, 0, 1)
BPF_STMT(BPF_RET+BPF_K, 1)
BPF_STMT(BPF_RET+BPF_K, -1)
```

Always Read/Written (cf. Figure 5.7)

```
BPF_STMT(BPF_MISC+BPF_LDC,BPF_GET_RWX)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x02, 0, 1)
BPF_STMT(BPF_RET+BPF_K, 1)
BPF_STMT(BPF_RET+BPF_K, -1)
```

Write, Read, Repeat (cf. Figure 5.8)

```
BPF_STMT(BPF_MISC+BPF_LDC,BPF_LOAD_STORED_2)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0, 2, 0)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 1, 7, 0)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 2, 0, 5)
```

```
BPF_STMT(BPF_MISC+BPF_LDC,BPF_GET_RWX)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x02, 0, 3)
BPF_STMT(BPF_LD+BPF_IMM,1)
BPF_STMT(BPF_MISC+BPF_STC,2)
BPF_STMT(BPF_RET+BPF_K, 1)
BPF_STMT(BPF_RET+BPF_K, -1)
BPF_STMT(BPF_MISC+BPF_LDC,BPF_GET_RWX)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x00, 0, 3)
BPF_STMT(BPF_LD+BPF_IMM,2)
BPF_STMT(BPF_MISC+BPF_STC,2)
BPF_STMT(BPF_RET+BPF_K, 1)
BPF_STMT(BPF_RET+BPF_K, -1)
```

Short List of Instructions (cf. Figure 5.9)

```
BPF_STMT(BPF_MISC+BPF_LDC,BPF_GET_EIP),
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x400513, 4, 0)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x40051e, 3, 0)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x400537, 2, 0)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x40053e, 1, 0)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x400552, 0, 1)
BPF_STMT(BPF_RET+BPF_K, 1)
BPF_STMT(BPF_RET+BPF_K, -1)
```

Monotonically Increasing/Decreasing (cf. Figure 5.10)

```
BPF_STMT(BPF_MISC+BPF_LDC,BPF_LOAD_STORED_1)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0, 0, 3)
BPF_STMT(BPF_MISC+BPF_LDC,BPF_GET_DATA_EA)
BPF_STMT(BPF_MISC+BPF_STC,1)
BPF_STMT(BPF_RET+BPF_K, 1)
BPF_STMT(BPF_MISC+BPF_TAX,0)
BPF_STMT(BPF_MISC+BPF_LDC,BPF_GET_DATA_EA)
BPF_JUMP(BPF_JMP+BPF_JGT+BPF_X, 0, 1, 0)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_X, 0, 1, 2)
```

```
BPF_STMT(BPF_MISC+BPF_STC,1)
BPF_STMT(BPF_RET+BPF_K, 1)
BPF_STMT(BPF_RET+BPF_K, -1)
```