Secure and Efficient In-Process Monitor and Multi-Variant Execution

SengMing Yeoh

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Binoy Ravindran, Chair
Ruslan Nikolaev
Haining Wang

December 11, 2020
Blacksburg, Virginia

Keywords: Multi-Variant Execution, In-Process Monitor, Memory Isolation

Secure and Efficient In-Process Monitor and Multi-Variant Execution

SengMing Yeoh

(ABSTRACT)

Control flow hijacking attacks such as Return Oriented Programming (ROP) and data oriented attacks like Data Oriented Programming (DOP) are problems still plaguing modern software today. While there have been many attempts at hardening software and protecting against these attacks, the heavy performance cost of running these defenses and intrusive modifications required has proven to be a barrier to adoption. In this work, we present Monguard, a high-performance hardware assisted in-process monitor protection system utilizing Intel Memory Protection Keys (MPK) to enforce execute-only memory, combined with code randomization and runtime binary patching to effectively protect and hide in-process monitors. Next, we introduce L-MVX, a flexible lightweight Multi-Variant Execution (MVX) system running in the in-process monitor system that aims to solve some of the performance problems of recent MVX defenses through selective program call graph protection and in-process monitoring, maintaining security guarantees either by breaking attacker assumptions or creating a scenario where a particular attack only works on a single variant.

Secure and Efficient In-Process Monitor and Multi-Variant Execution

SengMing Yeoh

(GENERAL AUDIENCE ABSTRACT)

Memory corruption attacks are still prevalent on modern software. While there have been many attempts at hardening software and preventing against these attacks, the heavy performance cost of running these defenses and intrusive modifications required have proven to be a barrier to adoption. In this work, we present Monguard, a high-performance hardware assisted in-process monitor protection system that provides an unintrusive and efficient way to defend against these attacks on monitor systems. We also introduce L-MVX, a flexible lightweight process monitoring engine running on Monguard that aims to solve some of the performance problems of recent monitor defenses.

Dedication

This thesis is dedicated to my wife, Joanne for her constant unwavering support

Acknowledgments

I would like to thank these amazing people for all the help, trust, and backing they've provided to me during my time at Virginia Tech:

Dr. Binoy Ravindran for taking a chance on my work and believing in me. Thank you for providing me with this opportunity to succeed and thank you for the guidance thus far - being part of your group has been the most pivotal point in my career. It has been a true honor to work under your wing.

Dr. Xiaoguang Wang for his constant help, advice, and guiding hand that has steered our research through rough and calm waters alike. Thank you for teaching me the best of researchers need to be both knowledgeable and humble at the same time.

Dr. Haining Wang for being a great committee member, providing guidance and assistance in the completion of this thesis.

Dr. Ruslan Nikolaev for agreeing to be on my examination committee and helping make this thesis a reality.

The Systems Software Research Group, for surrounding me with people way smarter than myself, so that I too may learn to be as ambitious and hungry for knowledge.

Last but not least, Virginia Tech for being a place I'm proud to call my second home.

Contents

List of Figures

List of Tables

List of Abbreviations

CFI Control-flow Integrity

DEP Data Execution Prevention

DOP Data Oriented Programming

ELF Executable and Linkable Format

GOT Global Offset Table

LLVM Low Level Virtual Machine

MVEE Multi-Variant Execution Environment

MVX Multi-Version Execution, used interchangeably with Multi-Variant Execution

PLT Procedure Linkage Table

PTE Page Table Entry

ROP Return Oriented Programming

TLB Translation Lookaside Buffer

TLS Thread Local Storage

Chapter 1

Introduction

In this digital era of cloud and ubiquitous computing where the internet and the applications running on or interacting with it have become a staple of everyday life the security of the data and the ability to mitigate impact of breaches or hijacks has become ever more cardinal to the safety and well-being of society as a whole. Yet attacks and data exfiltration or information leaks have become a norm as not only the amount of data at risk increases but the attack surface [51]. In particular, attacks on memory to directly hijack the control flow of programs [7, 9, 44, 49] or alter non-control flow data to modify the execution of the program [24] are still a very real and persistent threat [56]. For example, 3 of the Top 5 software weaknesses according to the MITRE 2020 CWE report are related to memory exploitation and corruption attacks such as out-of-bounds reads, writes, and memory buffer bounds violations like buffer overflows [41].

1.1 Motivation

Over the 30 year history of memory corruption attacks there have been many types of defenses proposed both in academic and commercial settings with the goal of defeating or hardening software against such memory attacks. Generally, these defenses can be broken down into three different classes [32]:

- Enforcement-based defenses
- Program monitors
- Diversity-based defenses

In our work we chose to focus on making the second and third class of defenses more robust, flexible, and lightweight. Program monitors and diversity-based defenses have some glaring weaknesses which pose barriers to greater adoption which we will discuss in the next sections.

1.1.1 Current Protection Domain Weaknesses

There is a great deal of work in the space of protection domains and memory isolation as it prevents untrusted components from accessing or modifying contents of security sensitive components of the system, such as cryptographic private keys [17]. Some techniques like Software Fault Isolation (SFI) [66] rely on compiler instrumentation of memory accesses while other approaches make use of hardware page protection to isolate memory without significant overhead while executing the protected component [5, 37]. In these systems switching control between components requires context switches to the kernel or changes in hypervisor mode. However, the fact that the monitor resides in a separate address space from the monitored code leads to unavoidable performance overheads when the untrusted code frequently traps to the monitor. Other technologies such as Trusted Execution Environments [25] have also been shown to have a high switching cost [30].

On the other hand, in-process monitors often have cheaper communication costs since no context switch is needed. However, protecting the monitor code itself becomes a new problem. For example, Shuffler [72] uses an in-process monitor to continuously randomize the application code location to defeat code reuse attacks. Some other research uses *protection*

domains within the target address space to isolate the monitor code. For example, segmentation in x86-32 CPU can be used to define logically isolated memory regions for sensitive data isolation [31, 73]. Unfortunately, the x86-64 architecture has mostly dropped support for segment limit in 64-bit mode. Finally, developing application reference monitors inside the kernel is a nontrivial work because it is very hard to avoid introducing unintended bugs to the system TCB [63].

1.1.2 Current Program Monitor Weaknesses

One of these challenges plagues program monitor systems and involves conflicting priorities of a defense - attempting to provide strong isolation between the monitor and the target program while maintaining low communication overheads. Many of these monitor systems utilize process-level isolation [45, 50, 63] while others use hypervisor-based isolation techniques [5, 29, 67] to separate the two entities. In the case of **Multi-Variant eXecution (MVX)** systems, several monitors make use of the `ptrace` interface to perform process-level isolation.

The downside of this is that the out-of-process monitors based on process-level isolation often come with overheads of up to 9x over an unprotected application when running commodity server applications like Nginx [63]. In addition to that, the monitors utilizing kernel-based [14, 70] or hybridized [63] approaches for isolation require intrusive modifications to the underlying kernel which poses a barrier to adoption of these defenses.

MVX systems also tend to suffer from false positives due to the isolation and differentiation between the variants [14]. Divergences between hosts when encountering non-attack payloads due to non-determinism can occur, requiring specialized techniques to deal with each class of false positive, such as result replication across variants. We believe reducing the protected

surface to only the necessary code accessing potential attacker input is sufficient to protect the entire program, reducing the need to handle false positives on benign code.

1.2 Thesis Contributions

The contributions of this thesis are the following:

- We introduce Monguard, a system in which a high-performance in-process monitor is efficiently isolated from the rest of the application leveraging Intel Memory Protection Key (MPK) technology, Procedure Linkage Table (PLT) call trampolines, and safe execution stack contexts.
- L-MVX, a practical use for the Monguard system in the form of a lightweight in-process Multi Variant Execution (MVX) monitor, allowing us to perform selective MVX protection at a PLT library call granularity on vulnerable sections of the program, further reducing the overhead of cross-variant monitor synchronization.
- Static analysis using LLVM compiler framework middle-end passes to further increase the flexibility and efficiency of L-MVX, allowing runtime memory diversification without breaking program semantics.

1.3 Thesis Organization

We organize the remainder of this thesis as follows: In Chapter 3 we go over the design and implementation for Monguard while in Chapter 4 talks about the design and implementation of the L-MVX system running on top of Monguard. Next, we showcase the evaluation results

from performance and security perspectives in Chapter 5. Lastly, we look into future work and potential improvements to our system and conclude the thesis in Chapter 6.

Chapter 2

Background and Related Work

This chapter aims to provide background information and context relevant to the rest of the thesis. First we talk about a few different Control Flow Hijacking Vulnerabilities in Section 2.1 followed by non-control data vulnerabilities in Section 2.2. We then talk about prior work done in similar threads to the contributions in this thesis which are capable of defeating these attacks in Section 2.3. Section 2.4 then discusses the process of dynamic linking, ELF binaries, and the `LD_PRELOAD` environment variable used to enable our work. Section 2.5 describes the Intel Memory Protection Keys hardware feature used to enable quick hardware accelerated switching of permissions, used in the Monguard system.

2.1 Control Flow Hijacking Vulnerabilities

Control flow hijacking vulnerabilities are a class of attacks allowing the attacker to take over the program control flow and perform unintended operations using the hijacked program. These types of attacks usually involve a memory corruption vulnerability such as a heap or stack buffer overflow or underflow that directly or indirectly enables the attacker to gain control of the program counter and redirect execution. The following is by no means an exhaustive list of current control flow vulnerabilities but rather provide a base with which to understand the threat model and how our defenses work against these attacks.

2.1.1 Buffer Overflows

Buffer overflows [2] are a critical vulnerability which serves as an entry point to an array of other attacks [7, 9, 44, 49, 58]. Buffer overflows occur when a buffer located on the stack is directly or indirectly written to by an attacker through a variety of libc calls (eg. `read`, `recv`, `strcpy` etc.). By not explicitly setting bounds checks in the program logic, the programmers can leave the program open to exploitation. When the attacker writes more than the buffer's capacity to the stack the remaining data spills over to the control data on the stack, the main target being the saved return address for the caller of that function. While the specific stack layout, local variables on the stack and their offsets vary based on architecture, the commonality is control data is stored on the stack as well.

Upon returning from that function, the overwritten return address is then popped from the stack to the program counter and the program resumes executing instructions from that location. In earlier exploits before the introduction of Data Execution Prevention (DEP) [15] the attacker could then redirect the program counter back to the stack which they had just overwritten, allowing them to perform code injection by including assembly instructions into the stack data being written. When DEP was introduced, this technique was rendered infeasible as the stack memory was marked as non-executable, preventing attacks from jumping back to it. This indirectly led to attackers attempting to jump to other locations in the program, reusing existing instructions or code in the target's address space to mount an attack.

2.1.2 Return-Oriented Programming Attacks

One of these attacks is known as **Return-Oriented-Programming (ROP)** [49]. Before we discuss ROP we first look into Return-to-libc (ret2libc) [58] attacks as they are connected,

where ROP is a more robust and expressive version of ret2libc. In a basic ret2libc attack on a 32-bit X86 system, the attacker performs a buffer overflow and gains control of the program counter and then proceeds to redirect execution of the program to a *code* region in the program, usually a the address of a libc function. For the rest of the discussion, we will be referring to Intel-based architectures. In 32-bit systems, since the call ABI [47] states that functions pass arguments on the stack, it is sufficient to include the injected arguments in the buffer that is being overflowed. A canonical way of performing this attack is by looking for the address of the `system()` libc call and the string `"/bin/sh"` and jumping to that libc call, opening a shell that can be used by the attacker. On 64-bit systems, this is more difficult as arguments are no longer passed on the stack but rather through registers [1].

This is where ROP comes into play; in ROP, the program address space is first scanned for *gadgets* which are chunks of instructions ending in an instruction serving to redirect the control flow to yet another location. Some examples of these gadgets can be seen in Figure 2.1. These gadgets end in either hardcoded immediate jumps, call instructions, or return instructions, enabling *gadget chaining* which means an attacker can string together a turing-complete string of gadgets as long as the redirection instruction jumps to another location within the attacker's control.

```
0x00000000000c32ff : add al, ch ; pop rdi ; cli ; call rdx
0x0000000000057124 : pop rdi ; retf
0x0000000000063347 : nop ; add qword ptr [rbp - 0x10], 1 ; jmp 0x63156
0x000000000006e864 : mov rsi, rcx ; call rax
```

Figure 2.1: Example ROP gadgets from the Nginx webserver

Analyzing the Nginx webserver's binary alone we were able to detect 30455 unique gadgets. On architectures like Intel x86-64 instructions have a variable length, meaning attacks are not limited to gadgets of a fixed size on an alignment, further increasing the number of gadget candidates. Utilizing these gadgets, an attacker can then set the `%rdi`, `%rsi`, `%rdx`,

`%rcx`, `%r8` and `%r9` registers which were needed to call the libc function and pass it arguments following the x86-64 ABI or perform other operations like stack pivoting which is especially useful when performing heap-based instead of stack-based buffer overflows.

2.2 Non-control Data Hijacking

Non-control data attacks involve attackers altering the program execution flow indirectly, without writing memory directly used in control transfer instructions [13]. This is also done without exiting the regular bounds of where the program can execute, unlike with ROP. By overflowing local variables and gaining control of non-control data pointers, attackers have shown they are able to perform privilege escalation or exfiltrate sensitive data [23]. Similar to ROP, these data-oriented attacks also take advantage of buffer overflow vulnerabilities which can corrupt memory.

Taking it a step further, Hu et al. have shown that in some programs, there exist gadget dispatcher loops which allows attackers to string together these *data-oriented gadgets* to form expressive turing-complete attack payloads in their **Data-Oriented Programming (DOP)** paper [24]. These gadgets also consist of short instruction sequences connected sequentially to perform logical, assignment, load, store, jump and even conditional jump operations. Once the attacker is able to corrupt local variables which gate the number of iterations a loop makes, they are able to consistently corrupt data and continuously inject/interact with their gadget dispatcher by corrupting the local variables to different data each iteration. The authors perform static analysis to locate and leverage these structures like the gadget dispatcher and the data-oriented gadgets. They show that purely course-grained memory randomization defenses such as Address Space Layout Randomization (ASLR) [3] are not sufficient to defeat DOP as they are able to find and leak the randomized location

of a server's OpenSSL private key.

2.3 Existing Mitigation Techniques

There have been a multitutde of mitigation techniques to defeat the attacks detailed in the previous sections [3, 10, 12, 72], we will focus on existing monitor isolation and Multi-Version-eXecution (MVX) techniques to contrast that with our work in this thesis.

2.3.1 Memory Isolation Techniques

An application can be split into different protection domains. This is especially useful for sensitive data protection (e.g., SSL key). Researchers have proposed using OS primitives [36], x86 segmentation [33], and even virtualization techniques [37] to protect sensitive data inside the address space. For example, **light-weight contexts (lwCs)** modifies the OS kernel to provide independent units of isolation within a process [36].

ARMlock [75] performs the opposite, opting to isolate untrusted modules within the application code to their own sandboxed domains. It functions by leveraging ARM's *memory domain* hardware feature support to create them. Memory accesses from within the sandbox are confined within them while system calls are interposed by the host where policies can be set.

Following this direction, recent work leverage Intel MPK to achieve sensitive data isolation with cheaper performance cost [21, 42, 61]. For example, **libmpk** is proposed as a library to virtualize the protection keys for the scalability problem [42]. **ERIM** further utilizes binary analysis/rewriting to prevent unintended sensitive MPK instructions from being maliciously used [61]. Although the MPK memory permission switch can be very efficient, MPK itself

does not guarantee the code page safety.

Other work like **IMIX** [20] introduce new proof-of-concept hardware primitives called `smov`, `sload` and `sstore` in order to isolate memory, running on Intel's Simulation and Analysis Engine. IMIX also requires kernel modifications in order to mark the relevant Page Table Entries (PTE) as secure or insecure. The authors argue that while other research in this domain sacrifices security in favor of performance, their approach enables strong and efficient in-process data isolation. It also avoids the need to perform bounds check instruction instrumentation for each memory access instruction.

2.3.2 MVX Techniques

The core concept behind Multi-Variant Execution (MVX) is that by having multiple diversified variants of a program, they ensure attack payloads will only work against a subset of the variants and any divergence in the flow of execution between variants will be picked up by a monitoring agent. The diversification can be in the form of stack growth direction [46], processor architecture [64, 68], or memory layout [6, 8, 62].

Ptrace (Cross-Process) based Monitors

`ptrace` [71] is a linux system call which allows allows a tracer process to observe and instrument changes into the execution of a tracee process by hooking and intercepting system calls and the `waitpid` polls the tracee for entry and exit of system calls. This makes it an ideal candidate for MVEE or MVX systems, allowing them to break the execution of tracee processes on syscalls and perform lockstep execution on the variants. These lockstep execution synchronization points (labelled Rendez-vous points in [62]) occur at syscall entry and exit points. Since the variants are executing on the same system, they have to perform I/O-

related system calls only once on one variant to avoid events like double writes which would change the semantics of the program, and can lead to errors. System calls with mutable results that vary every execution (non-determinism), or have side effects within the kernel are also replicated across variants in order to avoid false-positive detections. The primary disadvantage of ptrace-based systems is the number of data copies and context switches required between the kernel and both master and slave replicae.

In-process Monitors

There also exists some in-process monitors similar to our contribution in this thesis that exist in the process address space as a library. One such monitor is **VARAN** [22], which sits in userspace as a statically linked library but performs system call interception by rewriting the binary and patching system calls in the application binary to redirect the execution to dedicated handlers within a *coordinator* subsystem. The leader and follower variants synchronize and perform checks through a shared-memory ring buffer, replicating the data from the leader's execution for I/O calls in what is called an *event streaming* architecture.

This setup leads to very high throughput and low overheads of *1x* to *2x* even for target programs with a large number of I/O system calls as communication occurs via an efficient shared memory buffer, fewer context switches are required compared to `ptrace` MVX systems, and follower variants do not need to execute in lockstep with the leader variant. The disadvantage of not performing syscall synchronization on a single syscall basis is that there is a weakened security guarantee where an attacker can potentially exploit a memory corruption vulnerability and gadget chain before the followers detect the execution sequence divergence. This system also differs from our work in Chapter 4 as MVX execution and monitoring on VARAN occurs throughout the lifetime of the guarded program.

Kernel-based Monitors

Another approach is to implement the monitor in the kernel which can be faster than `ptrace` monitors as they require fewer context switches since the monitor resides within the kernel itself. Additionally, the monitor also must run in privileged mode, which can potentially introduce other vulnerabilities, thereby increasing the blast radius and compromising the integrity of the entire system.

N-Variant [14] is an example of a in-kernel monitor and is the pioneering paper in this field. It makes use of a heterogeneous memory space and instruction set tagging where the instructions are pre-pended with a tag bit to create diversity between variants along with the ability to create more than two variants. These type of monitors suffer from the complexity of requiring intrusive kernel modifications in order to be implemented.

HeterSec [68] creates a MVX system working across heterogeneous Instruction Set Architectures (ISA) on top of a framework which enables quick and efficient cross-node RDMA (Remote Direct Memory Access) communication and memory sharing. This is enabled by a multikernel distributed machine, synchronizing some HeterSec specific states and pages across nodes. The variance in ISA and stack layout between architectures guarantees that an attack payload working on a particular variant does not replicate its effects on another variant as ROP gadgets exist at different locations.

kMVX [76] is also an in-kernel monitor but focuses on protecting the kernel itself from information leak vulnerabilities. Kernel variants are generated on the same machine and run on virtualized or physical hardware, also utilizing the conventional leader-follower design of other MVX works. kMVX makes use of address space partitioning to generate different variants, logically halving the first virtual memory section to the leader and the next to the follower. Stack and heap variation techniques are also used to provide diversity in variants.

Hybrid Monitors

Lastly, we have a look at some examples of hybird monitors which use a combination of the previous methods to perform MVX. DMON [64] and ReMon [63] are examples of such monitors.

ReMon attempts to merge the advantages from these prior designs - secure MVX via a cross-process `ptrace` system, and an in-process monitor subsystem, merging the implementation from GHUMVEE [62] which ensures lockstep communication for monitored calls while an in-process monitor IP-MON performs result replication to different variants. The system also requires some in-kernel modifications in order to route the calls to the appropriate system depending if they are monitored or unmonitored calls though what is known as the In-kernel broker system.

DMON takes this one step further than ReMon by creating a distributed MVX system across discrete physical hosts running on different ISAs. DMON communicates across process nodes via what is calls a RC-COM, a reliable communication component used to exchange metadata. It introduces optimizations to the MVX system by utilizing similar techniques as previous work, categorizing syscalls into highly sensitive, moderately sensitive, and non-sensitive bins and protecting them appropriately be either performing lock-step execution, partially-checked execution, and unhindered execution respectively.

2.4 Dynamic Linking and `LD_PRELOAD`

To build programs, a compiler first needs to convert source code to object files. Each file then contains machine code instructions corresponding to the source program. In Unix-like systems the **Executable and Linkable Format (ELF)** is the de-facto file format for

binary executables, object files, and libraries running on these systems. As a result, there are three different variants of ELF files, each corresponding to the type of file it represents - executable, relocatable, and shared object ELF files.

There exist two different views of ELF files, the *link view* and the *execution view* as they convey different information depending on how the ELF file is being used. The ELF file contains *Program Headers* and *Section Headers* that describe this. When linking ELF files, the linker looks at the Section Header table which describes the different sections in the application. While loading ELF files, the loader then looks at the Program Header which describes the segments in the program which defines how the process address space looks like when loaded [34]. To promote code reusability static libraries contain commonly used functions. At link-time, static libraries with the ".a" extension (which internally comprise of a collection of raw object files) are linked to the program, enabling the linker to reuse the object files in the archive.

In order to address maintenance and resource problems with static libraries, modern systems make use of shared libraries and dynamic linking. The symbol binding and resolution of symbols contained in shared libraries occurs at runtime in contrast to static libraries and is performed by the dynamic linker, `ld.so` on Unix systems. At runtime, the dynamic linker usually searches libraries in the same order as is specified at static link time and uses the first definition of the symbol that it encounters.

This is where the `LD_PRELOAD` [19] trick comes into play; when invoking the program, passing in a library with a set of symbols we wish to preload tells the linker to bind these symbols before binding any other shared library symbol. **In our system we utilize this mechanism to perform two things - intercept specified libc calls by hijacking the specified symbols, and also to specify startup code to be run in the `.init` section to set up our protection and monitoring system before running the program.** The dynamic

linker binds these symbols to the program via a jump table known as the *Procedure Linkage Table* or PLT. Typical implementations of the dynamic linker support *lazy binding* which means that the location of the external symbols are not actually resolved in the PLT until the function is called for the first time. The `musl` [18] libc implementation we use in our system does not support lazy binding which means the symbols are bound on invocation time. This is to our benefit as will be discussed later as it enables patching operations to be run on initialization of the program.

2.5 Intel Memory Protection Keys (MPK)

Memory Protection Keys for Userspace (PKU) was introduced as an extension of the memory management architecture in Intel Xeon Scalable family (a.k.a Skylake-SP) [16]. It provides a mechanism to enforce page-granularity protection without modifying the page tables when an application updates the protection permission (PKEY). Therefore, it does not require TLB shoot downs and subsequent TLB misses. With MPK, bits 62:59 of each page table entry can be associated with one of the 16 available keys (PKEY). A new 32-bit thread-private protection key rights register for user pages (PKRU) was introduced to store the permissions of the 16 keys. For each key, there are two bits in the PKRU indicating the permissions for the thread currently running on that core: *write disabled* and *access disabled*.

To set/change permissions of a memory domain, an unprivileged instruction `wrpkru` can be used to update the PRKU register, which is a per-thread construct, enabling us to protect pages in different threads with different permissions. The memory permissions can be updated instantly and the permission bits of individual pages do not have to be updated when the access permission of a process or thread those pages are changed. Instead, these pages are initially associated with the specific keys and permission switching only needs to perform

a write operation to the PKRU register, saving us from having to enter the kernel.

This is different from the memory domain mechanism in ARM and PowerPC, where the kernel maintains the memory domain privilege [75]. Note that the memory key protection only works for memory *data accesses*. Interestingly, if the code pages are associated with an *access disabled* protection key, the code will be no longer read but still can be executed. This implements *the execute-only memory (XoM)* [4]. Monguard and by extension L-MVX leverage XoM to prevent trampoline code from leaking out the monitor location.

Chapter 3

Monguard Design

The first part of our contribution is Monguard, the protection system in which we can place in-process monitors efficiently isolated from the rest of the application utilizing Intel MPK to implement the execute-only memory, combined with code randomization to protect and hide the monitor. The system inserts instrumentation around sensitive instructions to prevent possible code reuse attacks.

In this chapter we cover the design aspects of Monguard, first going into the separation domains of the Monguard system in Section 3.2, followed by call gate setup and PLT patching in Section 3.3, followed by a description of the Monguard trampoline and its functions in Section 3.4, including stack-pivoting and multithreading support. Section 3.5 discusses how the Monguard system operates as a whole and sets the stage for how that will be utilized in Chapter 4.

3.1 Overview

Figure 3.1 shows a high-level overview of the application address space layout when running under Monguard [69]. Application code and data memory are separated from the monitor and shared libraries through a call gate which directly transfers the control from application code to the reference monitor. The call gate contains direct `jump` instructions to the monitor code. Monguard further marks the call gate code pages with *access disabled*. As a result,

attackers are unable to read the call gate code memory to locate the monitor in the address space, thereby preventing just-in-time payload generation attacks [52].

By doing so, Monguard ensures the call gate is the only legal entrance to invoke the reference monitor. Since the monitor was loaded at a random offset from the application code, it is difficult to locate the monitor address without having any direct pointer information. The only pointer information (monitor/library addresses) in the application context is embedded in the call gate as an immediate to the Monguard trampoline (within the `jump` instructions). While an attacker may try to brute-force the address space in order to modify the monitor credential data (e.g. change the result of a monitor request), those attempts to write to data from the potentially malicious application code will be prevented by MPK protection.

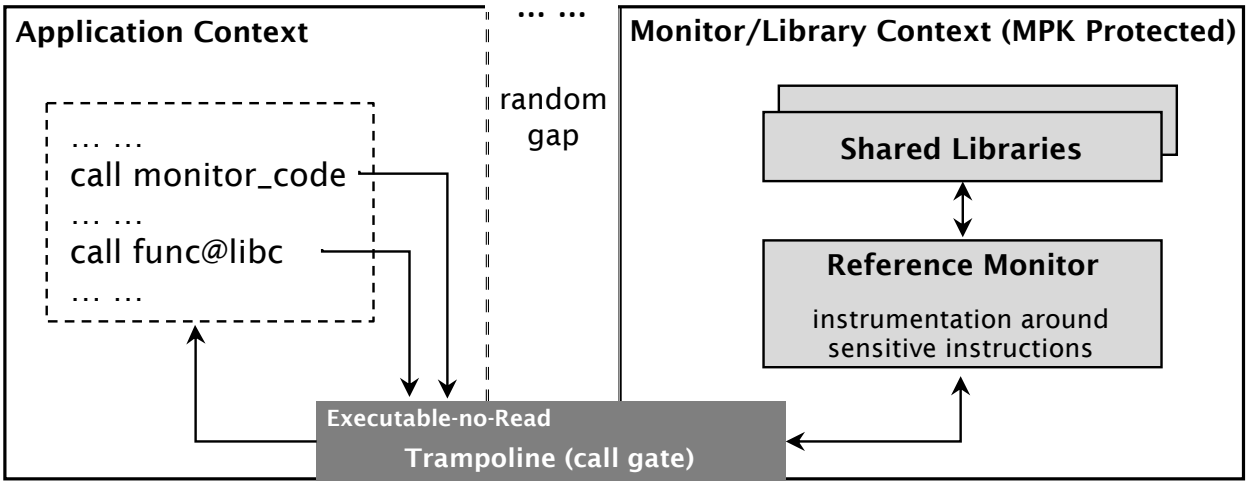

Figure 3.1: System overview of Monguard with application context and the monitor (shared libraries) context. The call gate memory is executable but not readable.

3.2 Monguard Context Separation

Monguard maintains separate contexts; namely the **"safe context"** in which the reference monitor and certain shared libraries such as musl libc live and the application context which

we also refer to as the **"unsafe context"** in which the untrusted application code and data reside. The Monguard trampoline serves as the only legitimate entrance and exit to and from the safe context. Any attempts to access data across contexts from the unsafe application context to the safe monitor context without first passing through the call gate results in an MPK protection fault as all the pages within the safe context are assigned their respective PKEYs.

PKEY 1 is assigned to all code pages, with the respective register of that key disallowing reads and writes to those pages, causing these pages to have execute-only permissions. A second protection key, PKEY 2 is associated with the data pages of the components in the safe context, where this PKEY is what will enforce gate access between contexts and will be switched by the trampoline according to the context currently being executed. PKEY 2's permissions will be set to *access disabled* when the application is executing in the unsafe context and cleared by the trampoline before entry to the safe context. Similarly *access disabled* is set again upon exiting the safe context.

3.3 Call Gate Setup and PLT Patching

The Monguard call gate is a `jmp` instruction followed by the memory permission (PKRU) update. The `jmp` instruction is originally from the PLT code, which transfers control to the monitor. Monguard leverages the structure of the shared library to hide the location of the monitor. Specifically, the compiler generates the PLT/GOT for each external library function (e.g., `printf` in libc). Each PLT entry contains an indirect jump with the destination address stored in GOT (the `.got.plt` section in ELF binaries). The loader resolves the external function address and fills in the corresponding GOT entry in what is known as on-demand symbol resolution (lazy-binding). However, there is a subtle security issue - the

GOT contains the the monitor and libraries addresses which can then be leaked.

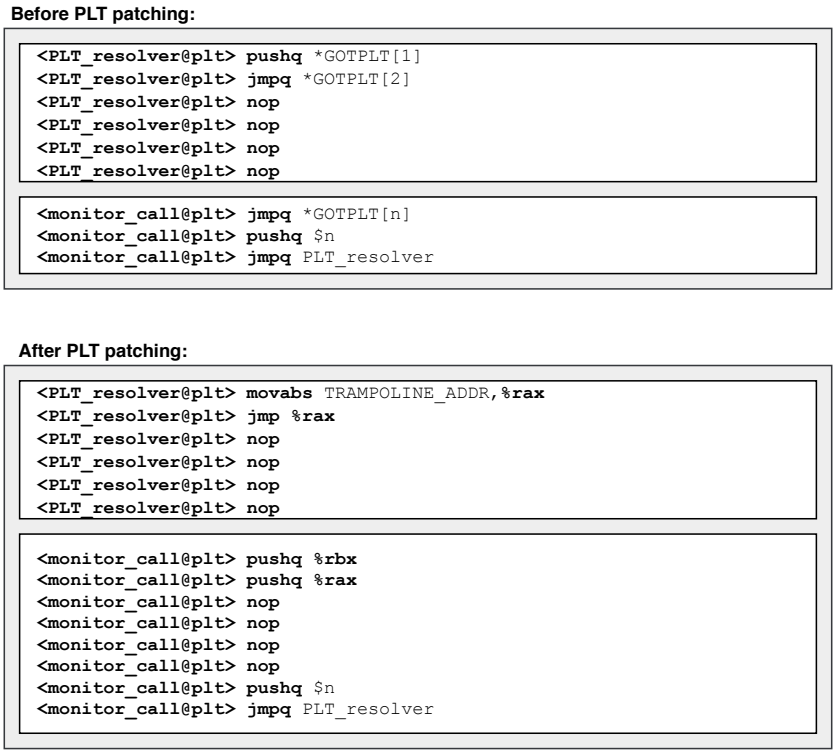

Figure 3.2: Original PLT Resolver and Call slot patched

The premise of Monguard relies on the reference monitor being well hidden against potential attackers. Since we have preloaded the libc calls with our call gates, the `.got.plt` slots now contain pointers to the hidden reference monitor. To prevent the program from leaking information about the location of the reference monitor in the address space, we take advantage of musl-libc's lack of lazy binding support. By the time the constructor of Monguard is called, the `jmp` instruction in the PLT has already been resolved to reference its respective slot in the `.got.plt` section containing the address of the call targets in the shared library.

To solve this issue, we patch the `.plt` section a shown in Figure 3.2 where both the lazy binding resolver's location and an individual PLT slot corresponding to each libc call is patched. We also make use of existing instructions within the PLT slots to minimize the amount required to be modified. Subsequently, we apply the execute-only memory protection

to the PLT code pages. This patching is performed on program startup in the Monguard libc constructor, corresponding to the `.init` function of the Monguard library.

Monguard automatically intercepts external library calls (e.g., libc calls). Security researchers can implement other monitor calls, for example to check memory integrity [31] or transform the code [72]. To write a monitor call, developers have to declare the monitor call function type and compile/link the monitor call skeleton (an empty function) to the application binary.

We walk through the execution steps of a libc monitor call through the patched PLT:

1. Starting from the PLT slot for a particular libc call (labelled as monitor_call in this example) we store the values of `%rbx` and `%rax` first.

2. Next, we slide down the `nop` instructions and land on the existing `pushq` and `jmpq` instructions which redirect execution to the PLT resolver, sitting at slot 0 of the PLT section.

3. In the PLT resolver the immediate MPK trampoline address is jumped to.

4. We hijack the PLT resolver's location and use it to redirect all libc calls to an a trampoline whose function is as a call gate to switch the MPK protection bit by writing to the `wrpkru` register, create the illusion that there was no Monguard trampoline call, and create a Monguard trampoline stack.

In the next section we look at the details of the Monguard trampoline and how it achieves this.

3.4 Monguard Trampoline Anatomy

Once the application invokes a monitor call, the patched PLT redirects the control to the monitor code. The monitor clears the `%rax` value to prevent any potential monitor address leakage. Next, the monitor deactivates the PKEY protection for monitor data and will reactivate it when leaving the monitor. An extra benefit of using the trampoline `jmp` instruction to update MPK protection is that we can hide the sensitive `wrpkru` instructions from the application code. For those unintended `wrpkru` instructions occurred in the application code, we could adopt similar techniques used in ERIM [61] or Hodor [21]. For example, using binary rewrite to replace the unintended `wrpkru` instructions with semantically same instructions [61] , or using the debug register to monitor the unintended `wrpkru` use [21]. In order to prevent internal libc functions from calling other libc functions through the PLT and going through the call gate again, we passed the *-Bsymbolic* flag to the linker when building musl-libc, thereby preventing symbol inter-positioning.

Listing 3.1 shows the pseudo code of a Monguard trampoline. The monitor defines a global dummy variable (line 1 in Listing 3.1). We instrument a dummy variable write before each indirect control transfer instruction, preventing a powerful attacker from performing a ROP attack into the monitor code and hijacking the control flow back to application code. The monitor DEACTIVATE is a macro of inline assembly removing the PKEY data access protection. The Monguard trampoline then moves `%rsp` to the Monguard trampoline stack and jumps to the corresponding GOT entry. Recall that this GOT entry has already been preloaded with `LD_PRELOAD` thus the call is redirected to the equivalent symbol in the in-process monitor.

```
1  int canary = 0; // Barrier variable
2  int monguard_trampoline @Ref_Monitor() {
3          /* Clear %rax used in .plt trampoline */
4          asm("xor %rax,%rax");
5          DEACTIVATE(); // Disable data protection
6
7          /* Swap out current application stack with per-thread tls safestack */
8          swap_to_safe_stack();
9
10         /* Push arguments 10, 9, 8, 7 onto safestack */
11         build_safestack_layout();
12
13         /* Reference monitor implementation */
14         real_monitor_code();
15
16         /* Swap back to application stack */
17         swap_back_to_unsafe_stack();
18
19         /* Touch monitor data */
20         asm("mov $0x0, %0", = r(canary));
21         ACTIVATE(); // Re-enable protection
22         /* Return to application */
23         return retval;
24 }
```

Listing 3.1: Mongard trampoline

After completing the real monitor code on line 14, the Monguard trampoline then swaps `%rsp` back to the application's unsafe stack. Finally, the monitor re-enables the protection by using ACTIVATE macro. The ACTIVATE macro's implementation is very similar to the

call gate used by ERIM [61]. Specifically, it enables the MPK protection by updating the PKRU and checks the `eax` value after the `wrpkru` instruction to prevent the control flow hijack breaking the integrity of the gate code.

3.4.1 Stack Pivoting to Monguard trampoline stack

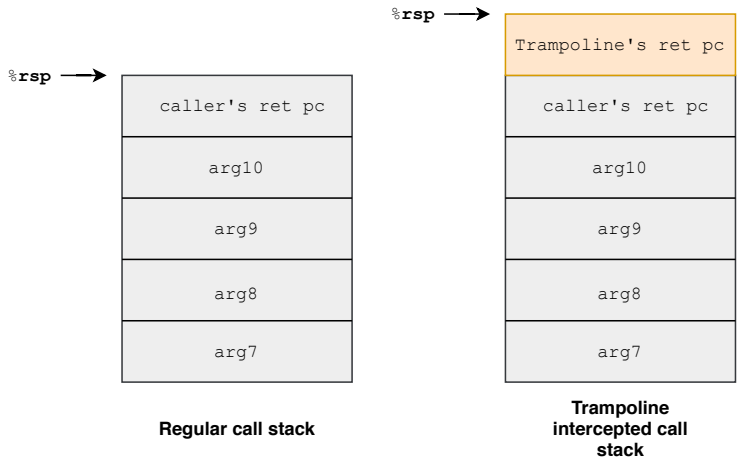

Figure 3.3: Stack layout seen by monitor/libc function without and with the trampoline interception

The stack pivot to a Monguard trampoline stack is required for several reasons, the primary one being the ability to **maintain strong isolation between the secure reference monitor and the unsecured application stack**.

In addition, stack pivoting is required to solve the problem that arises when an intercepted libc call has more than 6 arguments. The x86_64 function call convention [40] dictates that integer arguments 1-6 of a function call are passed through registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` and `%r9`. Any additional arguments are pushed onto the stack. Further, for variadic functions the number of float arguments is also stored in `%rax` and call-time. These rules pose a problem for our trampoline as the libc or monitor (depending if that particular symbol is preloaded) function instructions do not take into account the added Monguard trampoline

intercepting its execution.

As seen in Figure 3.3, the addition of a `callq` instruction to the reference monitor (corresponding to the pseudocode at line 14 of Listing 3.1) results in the return address of the trampoline being pushed onto the stack - causing any `%rsp` relative addressing within the function code to no longer be semantically correct. Replacing the `callq` instruction with a `jmpq` instruction is not an option as we need the reference monitor function to return to the Monguard trampoline so it can reactivate MPK protections and swap back to the unsafe stack.

Critically, `%rbx` also needs to be saved by the callee (in this case the Monguard trampoline), limiting the number of registers we are free to use in the Monguard trampoline to perform the stack pivot and MPK pkru bit set/reset.

We resolve this issue by leveraging the Monguard trampoline stack to rebuild the stack in an MPK secured location, replacing the return pointer to the caller with a return pointer to the trampoline, thereby making the interception by Monguard completely transparent to the reference monitor and the subsequent libc call. `%rbx` is also saved onto the unsafe stack and will be popped upon returning to the Monguard trampoline. The `%rax` value on entering the trampoline is restored and passed into the call to the reference monitor so variadic function calls into the reference monitor remain unbroken.

3.4.2 Handling Multithreading

In order to support multithreaded applications, the Monguard trampoline stack memory area and unsafe stack pointer which is used to store and restore the address of the application stack before and after entering the monitor are created as **Thread Local Storage (TLS)** variables within the Monguard library's address space. This allows each newly-created thread

to have its own Monguard trampoline stack to jump to when entering the monitor as shown in Figure 3.4.

The Monguard trampoline stack is also guarded with Intel MPK to prevent attackers from reading or writing to it while executing the application code.

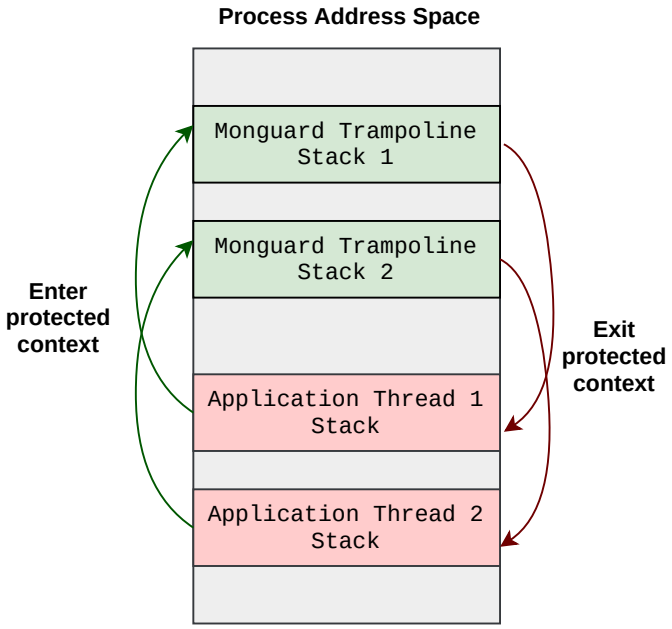

Figure 3.4: Individual per-thread Monguard trampoline stacks

3.5 Putting It All Together

Figure 3.5 shows the execution flow of an function call to a call being monitored by Monguard, putting together all the elements discussed in the previous subsections. The application code calls a monitored call which then calls to the patched PLT which subsequently sets up a jump to the MPK-protected Monguard trampoline which deactivates MPK protection by writing to the PKRU register. Next, the stack is switched after Monguard trampoline stack assembly, the correct overridden symbol is called and execution is transferred to the reference monitor

which upon completing, returns to the trampoline. The Monguard trampoline switches the stack back to the unsafe stack and reactivates MPK protection before returning to application code.

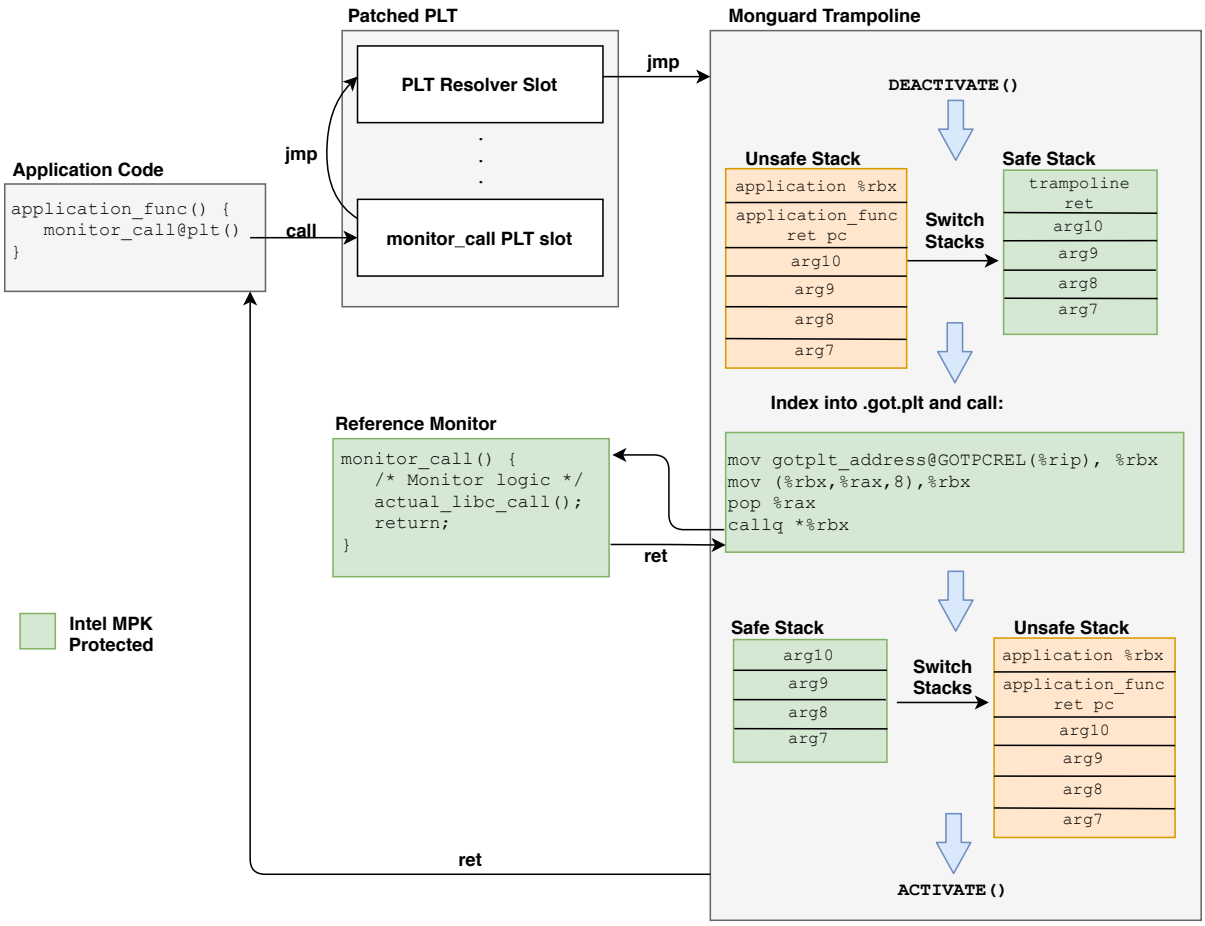

Figure 3.5: Execution flow of a monitor function call through Monguard

3.5.1 Case Study - An in-process Multi-Variant-Execution (MVX) Monitor

Now that we have a highly performant way of creating and isolating the in-process monitor for a guarded process, we seek to utilize this hardware-assisted isolation. As mentioned in

the background, multi-version or multi-variant execution has proven to be an effective way to defeat control-flow hijacking attacks due to application diversity, and in our case, memory layout asymmetry - the assumption that a single attack payload is only guaranteed to work on one particular variant at a time and will cause a divergence [14].

Figure 3.6 shows a Monguard-protected system and the MVX Monitor's placement within the system. The monitor and libc context is protected using the MPK callgate and two variants run concurrently as separate processes. The primary variant (variant 1) serves as the point of truth and source of data for I/O operations while the secondary variant (variant 2) duplicates execution of the first variant for intrusion detection purposes. Variant 1 and Variant 2 are loaded into non-overlapping addresses and communication between them is synchronized via a shared-memory IPC.

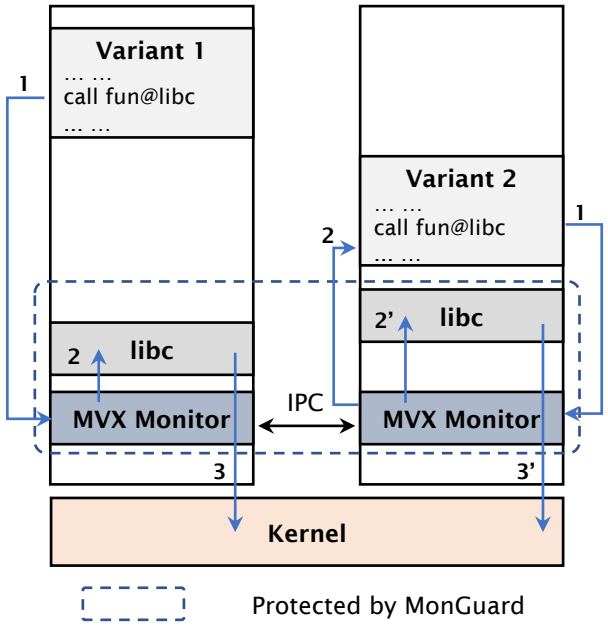

Figure 3.6: In-process MVX monitor protection with MonGuard

Chapter 4

L-MVX Design

In this chapter we introduce a working prototype of a novel **Multi-Variant Execution Environment (MVXEE)** called **L-MVX (Lightweight-Multi-Variant Execution)** which utilizes the Intel MPK-based hardware isolation to extend the Trusted Computing Base (TCB) to an MPK guarded in-process libc call interception library to create a lightweight and selective MVX protection mechanism. We make use of the `LD_PRELOAD` dynamic linker trick described in Section 2.4 of this thesis to intercept libc calls and use them as synchronization points between the variants, comparing arguments and return values, as well as emulating buffers for IO-related libc calls and thread local values such as *errno*.

Section 4.1 goes over a quick overview of the system, followed by a the setup steps required to run the L-MVX system in Section 4.2. We also go over the lifecycle and execution sequence of a program protected by L-MVX in Section 4.3, followed by an explanation on how MVX is performed on libc calls and synchronized across variants in Section 4.4 and how functions to guard are selected in Section 4.5. Finally, we talk about the potential pitfalls of performing runtime memory relocation and some approaches to solve it in Section 4.6 and 4.7.

4.1 Overview

Existing MVX techniques using out-of-process monitor designs have shown to incur heavy communication overheads while context switching between the monitor process and the

target process as described in Section 1.1.2. Meanwhile, these monitor designs also enforce protection on the entire program execution lifetime. We contend there is no need to protect the entire program but rather specific subtrees of the target program's call graph, specifically I/O handling code, as shown in Figure 4.1. An alternate variant of the program with a non-overlapping address space, listed as Variant 2 is spawned at the start of the protected region and destroyed on exiting it. Within the protected region we perform libc call synchronization and checking through a secure IPC protected using Monguard. By doing this we avoid the communication overheads of protecting non-vulnerable regions and further reduce the rate of false-positive execution divergence detection. Keeping the monitor within the process further reduces the overheads that we do pay while protecting a vulnerable region as there is no communication or lockstep execution between monitor process and target process outside of the vulnerable region. Note that L-MVX is also capable of protecting arbitrary regions in code and can be extended to protect the entire program execution tree as required and is not solely restricted to protecting vulnerable regions.

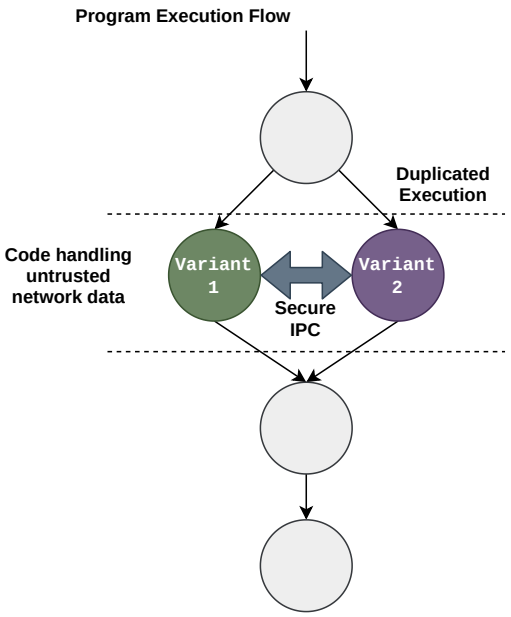

Figure 4.1: L-MVX Duplicated Execution only for input handling code

4.2 L-MVX Setup and Flexibility

In order to set-up L-MVX, we require the target application be linked against a custom library providing function stubs which will then be preloaded at load time to point to the `LD_PRELOAD`ed MVX Monitor library.

```
int main(void) {
        lmvx_init();
        /* Unprotected area */
        lmvx_start("protected_func", 2, arg1, arg2);
        protected_func(arg1, arg2);
        lmvx_end();
        /* Unprotected area */
        return 0;
}
```

Listing 4.1: Example of a function protected by L-MVX

Listing 4.1 demonstrates the order which these calls are called with respect to the protected application. **Note that the actual implementation of these calls is contained in the Monguard protected monitor library and not in the initially linked library.** The primary advantage of this setup is the ability to shrink or grow the protected region to encompass only vulnerable regions in the program at a function level granularity.

The function of `lmvx_init()` is to ultimately call the syscall `SYS_pkey_mprotect` on the Virtual Memory Areas (VMA) for both libc and the monitor library to associate the allocated *pkey* with the corresponding pages.

`lmvx_start()` triggers the cloning of the process, updating of shared values, creation of the shared memory IPC and makes variant 2 jump to the correct function `.text` address. This is the most critical function and we'll go into this process in detail in the next subsection.

The role of `lmvx_end()` is to synchronize the lock-step libc checks and wait for checks to complete before allowing variant 1 to continue with its execution normally after exiting the vulnerable protected function.

We also require the user to run a script that analyzes the binary using the tools from GNU binutils and dumps information from the section headers regarding the start address and size of each section in the ELF binary, along with symbol table symbol to address mappings into a temporary file which is then read by the L-MVX monitor.

Enabling and disabling our current implementation of L-MVX does require access to source code and the compilation and link process of the application, but only to inject the aforementioned function stubs. This can also be achieved by lifting the application binary using binary lifting tools such as McSema [57] to *LLVM Intermediate Representation (IR)* before modifying/patching the LLVM IR with inserted function call instructions at the correct locations and re-linking it to call our dedicated L-MVX library functions.

4.3 L-MVX Lifecycle

Before running the application with L-MVX, the user will need to run the aforementioned script to create a file in the `/tmp` filesystem containing info about the start offsets and size of the `.text`, `.data`, `.bss`, `.plt`, and `.gotplt` sections in the application. We also save the symbol table information so we have the addresses of functions. This mapping will then be used to ascertain the offset of a selected symbol and instruct Variant 2 to begin execution from that point.

When an application is run with L-MVX, the L-MVX monitor library which is preloaded using the `LD_PRELOAD` linker environment variable first runs its constructor function `setup_lmvx()`.

This constructor function is then called by the libc dynamic loader/linker and performs setup for L-MVX, specifically:

- Storing the original libc symbol addresses so we can use them internally within the library without intercepting ourselves.
- Initializing the loader subsystem within the library which is responsible for loading Variant 2 into memory. This reads the information from the previously created `/tmp` filesystem file, reads the proc info from `/proc/self/maps` and patches the loaded `.plt` of the running process with the MPK trampoline previously detailed in the Monguard Chapter 3 of this thesis.
- Setting up shared-memory IPC including the mutexes, condition variables, and libc call emulation/call data.

Figure 4.2 shows the call graph of a program protected by L-MVX. By placing the `lmvx_start()` and `lmvx_end()` calls on either side of the call to `function2()`, its entire subtree of the call graph is protected by L-MVX. `main`, `function1()` and `function3()` (and by extension its own subnodes) do not have their execution duplicated with Variant 2, only `function2()` and its child nodes run in lock-step with Variant 2.

4.4 LibC Synchronization and Check Points

L-MVX synchronizes data between both variants via a secure shared memory IPC. The L-MVX system checks several parameters for consistency across variants.

- Function names that are being called.

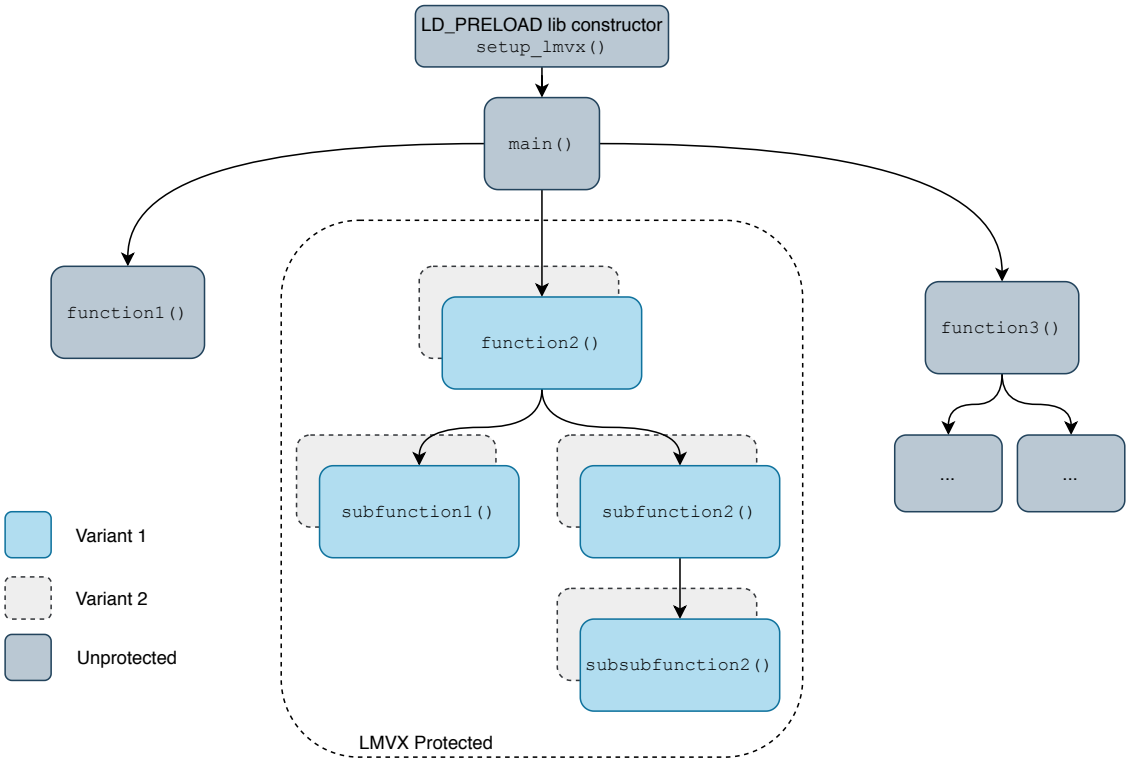

Figure 4.2: Callgraph of a program protected by L-MVX

- Return values from each libc call.
- Argument values.

We currently do not support checking of pointer arguments as the differing memory layouts of both variants will invariably result in different pointer argument values. On top of that, L-MVX also needs to simulate certain values for certain libc calls, specifically those which read/write and perform I/O operations as sharing the same filesystem results in a situation where 1 variant may read and clear a buffer or handle and clear an event (i.e in the case of `epoll_wait()`) resulting in the next variant arriving at that libc call no longer seeing that data and subsequently causing a divergence in execution.

As is mentioned in the Orchestra [45] paper, other libc calls which require simulation include time-sensitive libc calls (i.e `gettimeofday()` and `localtime_r()`) and reads of

`/dev/urandom` and equivalents.

Table 4.1 shows the libc calls with these emulation requirements which we supported in order to run L-MVX on the applications showcased in the evaluation section.

All 3 categories of libc calls needing emulation on Variant 2 also require `errno` emulation in order to correctly give Variant 2 accurate information regarding the error status of the libc call. The first category of libc calls requiring emulation only require emulation of function return values on top of errno. These libc functions do not write to any application buffers passed in through arguments. The next category of libc calls write data to their corresponding arguments, passing data back to the application through pointers in the input parameters. When Variant 1 actually runs the libc call, it copies over this data to Variant 2 through the IPC, allowing Variant 2 to skip execution of that libc call again.

Finally, there are a few libc calls which require special handling. `ioctl` has variable arguments depending on what request and device driver it is sending commands to. This results in L-MVX being unable to detect if any of these arguments are pointer arguments and if emulation is required. In our approach we noticed that the only call made to `ioctl` by our test applications was made in the form where the third argument was a pointer which needed to be emulated and thus only handled that case. One way to handle this in a more robust and generalize-able manner would be to only emulate the input argument if it is a pointer and falls within the process' address space.

`epoll_wait` and `epoll_pwait` are slightly different because whether or not they need to be emulated depends on the mode of `epoll_data` being used. `epoll_data` is a union that can be a file descriptor, 32-bit value, 64-bit value, or a void pointer to data defined by the application. The difficulty arises when the union represents a pointer value as this will need to be emulated on Variant 2. We solved this by checking if the value falls within the process'

	Libc Function Names
Libc calls only requiring return value emulation	`open, close, shutdown, writev, write, epoll_ctl, setsockopt`
Libc calls requiring return value and argument buffer emulation	`sendfile, stat, read, gettimeofday fstat, accept4, recv, getsockopt, localtime_r`
Libc calls requiring special emulation	`ioctl, epoll_wait, epoll_pwait`

Table 4.1: I/O related libc functions with emulation requirements

address space, as with `ioctl`.

4.5 Taint Analysis

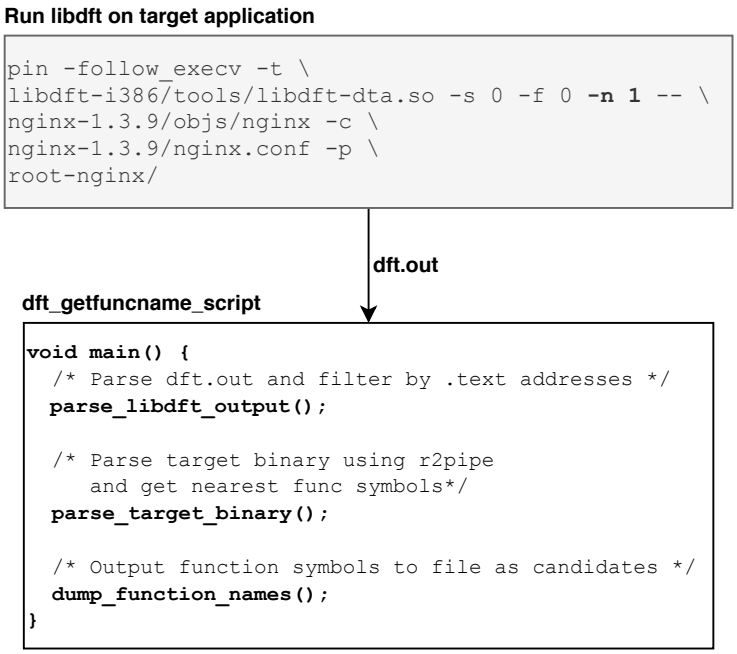

Figure 4.3: Taint Analysis Workflow

In order to make use of L-MVX without compromising on the security of the application from a remote attacker's standpoint, we need to select vulnerable functions in a program to protect with L-MVX. The selection is done by performing **taint analysis** using `libdft` [27]

a dynamic data flow tracking library. Network-related memory is tagged during program execution and is tracked as it is copied and altered by instructions in our target programs. This taint data denoting memory that read or wrote to tagged network memory is output to a file.

Next, a script is run to get the functions in which those instructions reside. We make use of the r2pipe [43] in our script to send the command `fd` to get the closest symbol to the instruction to get the function candidates to protect with L-MVX. This workflow is shown in Figure 4.3.

For the nginx webserver, some of the functions which handle tainted network data directly are as follows. These will be candidates to protect with L-MVX.

- `ngx_http_handler`
- `ngx_http_header_filter`
- `ngx_http_parse_request_line`
- `ngx_http_index_handler`
- `ngx_http_process_request_line`
- `ngx_http_validate_host`

4.6 Pointer Scan and Relocation

Moving the entire process address space to create a diversified execution across variants while the program is running comes with challenges. These challenges are similar to what is faced by RuntimeASLR [39], but to a lesser extent. Figure 4.4 shows the issue with relocating

the process address space. Upon entering the protection zone (executing `lmvx_start()`) the L-MVX engine jumps to a cloned and shifted version of the process address space, sans the heap.

By linking the program as a **Position Independent Executable (PIE)** the code within the executable's address space accesses its own data by using IP-relative addressing. This allows for relocation of the entire process address space, which is what **Address Space Layout Randomization (ASLR)** [3] relies on. This case is depicted with the blue arrow in Figure 4.4 where the `.text` section is still able to legally access its `.data`. However, since we are performing L-MVX in the middle of executing the program, there are pointers in global data and the heap which contain addresses corresponding to the now-unmapped original location of the process. These are shown as red arrows in the figure.

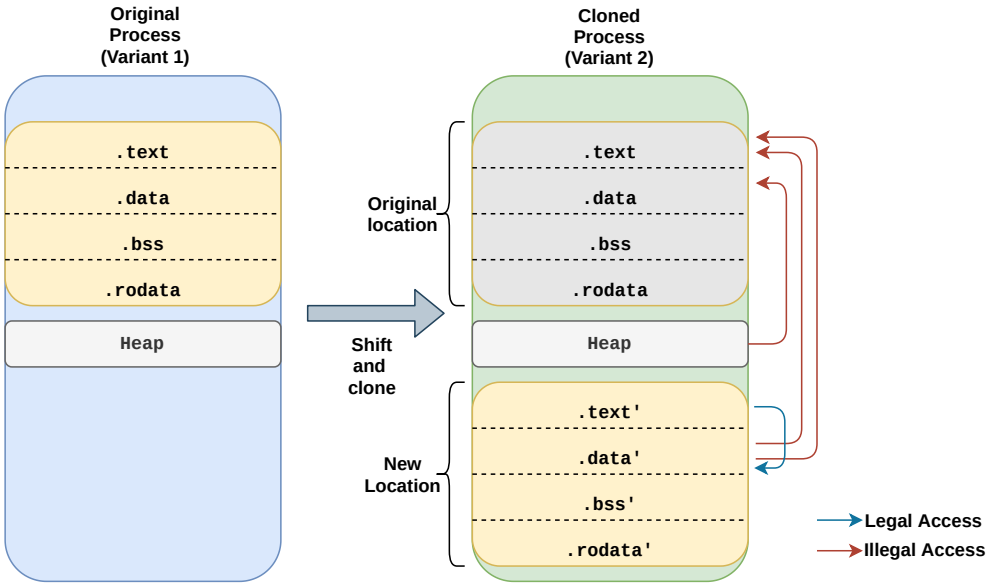

Figure 4.4: Pointer relocation problem

As a result, we have to scan the `.data` `.bss` and heap for function pointers to the old `.text` location so execution doesn't attempt to jump back to the old process location. We also need to scan for data pointers pointing at the old `.data` and `.bss` locations to prevent execution

of Variant 2 from attempting to access illegal memory. Once we have the addresses of these pointers, we update the pointers with the appropriate offset to ensure they point to the shifted address space.

A weakness of this approach is the scan and update process occurs at runtime on execution of `lmvx_start()`, causing a significant overhead which can impact the application if the protected region is entered and exited within the application's control loop cycle as the overhead is multiplied by the number of times the control loop is executed.

4.7 Compiler Techniques

To solve this issue we attempt to break the pointer problem into several components and solve each of them by performing static analysis on the target program. If we can find out the addresses in the program that contain pointers that require updating, we can update only those pointers without having to walk through memory, saving a substantial amount of time. For the remainder of the discussion we will consider the moved `.data`, `.bss`, and `.text` as `.data'`, `.bss'`, and `.text'`.

1. **Global' to global pointers**: This is the case where pointers in global variables at the new location (`.data'`, `.bss'`) still reference data at the original location (`.data`, `.bss`). Global' refers to the set of both `.data'` and `.bss'`.

2. **Heap to global pointers**: The case where the cloned heap still has pointers to global data. The solution for this case has not been implemented.

3. **Code pointers in heap and global data**: The case where code pointers to `.text` exist in Heap, `.data'` and `.bss'`. The solution for this case has not been implemented.

4.7.1 Global' to global pointers

We attempt to solve this through static analysis using the **Low Level Virtual Machine (LLVM)** [38] framework. In this case, we have access to the application source code and compile it down to LLVM bitcode. In order to analyze the program as a whole, we then merge and link the individual bitcode files using one of the tools included in the LLVM framework: `llvm-link`.

```
MVXAA::runOnModule(Module &M) {
    /* Run SVF on module */
    runWpaAnalysisOnModule(M);

    /* Create call graph for function subtree */
    CallGraph *CG = createCallGraph("guarded_function_symbol");

    /* Visit each function in CG */
    foreach function in CG:
        this.visit(function);

    dump_global_names();
}
```

Listing 4.2: Visiting load instructions in a function in the guarded call tree.

Next, we run our custom LLVM middle-end pass on the merged bitcode file, feeding the pass the name of the function we intend to guard. We are able to use an LLVM Module pass to treat the entire program as a single module as it has been merged. As shown in Listing 4.2, the pass first uses the **Static Value Flow Analysis (SVF)** [53, 54] framework to analyze the target program and run a **Whole Program Pointer Analysis (WPA)** pass on the program to perform pointer field-sensitive alias analysis.

The pass then uses the function name to create a call graph of the function including all its callees and iterates through them, running the pseudocode shown in Listing 4.3 on each function. Before going into the details of the algorithm run on each function, we explore the structure of a `load` instruction in LLVM, and the `GetElementPtr` (GEP) instruction that frequently goes along with it, used to index into global structs as seen in Figure 4.5.

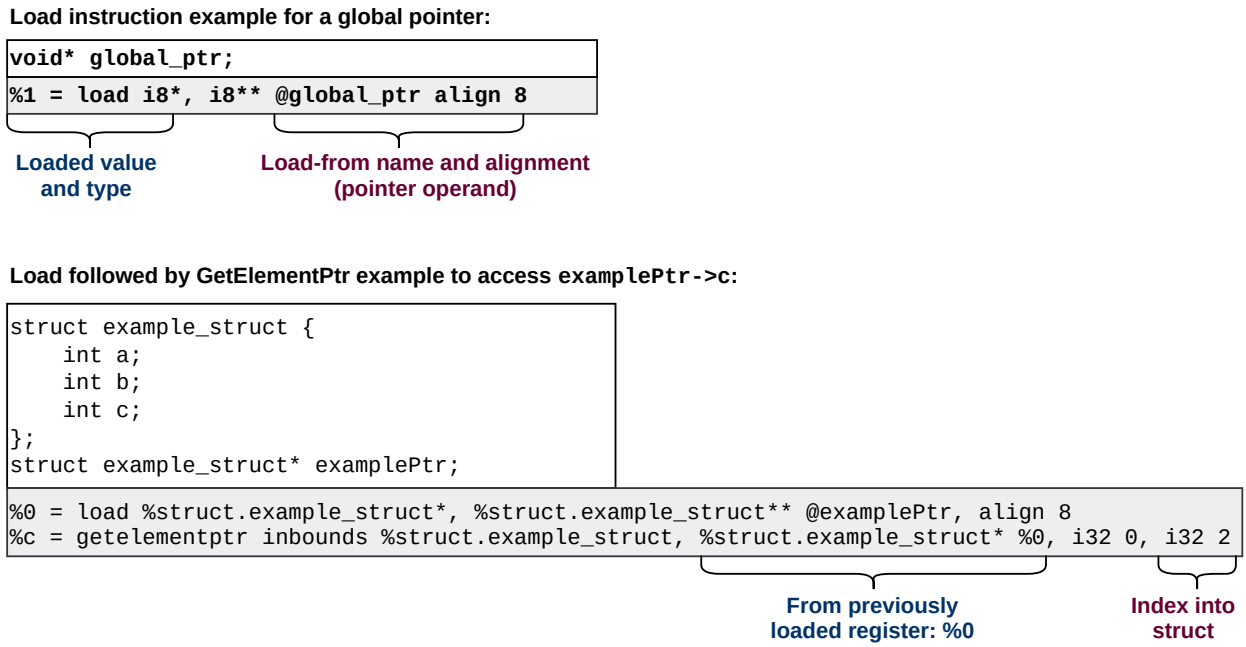

Figure 4.5: Load and GEP instructions

In LLVM any global memory access is treated as a pointer to that global memory area. This means any *use* of a global variable is invariably preceded by a `load` instruction to load that value from the global memory area. In Figure 4.5 we see that the value from the pointer operand `global_ptr`(which is a double-pointer due to LLVM's treatment of all globals as pointers) is loaded into an LLVM register `%1`. The second example demonstrates a GEP instruction used to index into a similarly-loaded register `%0` to access the third element in the struct.

Bearing this in mind, we now take a look at the algorithm in Listing 4.3. Each guarded

function is visited and the pointer operand of the `load` instruction is checked against the alias database to see if it aliases any globals in the module. If it does alias a global, this means it is either a global or is a pointer previously transferred from another global (by casting, or through a GEP instruction). Next we look at what is loaded from that global and check if it too aliases a global (meaning it too is a global) and is a pointer type. Finally, if the pointer operand of the load instruction is in the global set (meaning we're loading directly from the global) we add the symbol of that global to our output set with a zero offset as this value isn't a global struct element. However, if the load instruction only aliases the global and isn't part of the global set and is a GEP instruction, this means what we're loading now must have been loaded from a global struct in a previous `load` instruction. We go up the load + GEP chain and get the base load address, along with the offset into the struct.

```
MVXAA::visit(Function &f) {
    foreach load instruction in f:
        if (load.pointerOperand aliases any globals):
            if (load.loadedVal is pointer type and aliases a global):
                if (load.pointerOperand is in the global set):
                    Add to zero offset set, this means it is a direct load
                    without GEP instruction.
                else(if we only alias the global, and are a GEP instruction):
                    Find base load of GEP and get global symbol name.
                    Get offset from GEP Instruction.
}
```

Listing 4.3: Visiting load instructions in a function in the guarded call tree.

Finally we output the symbol names and offsets (as is the case for struct members) of this set of global' pointer variables which have pointers to other globals to a file. We can then use a script utilizing the aforementioned *r2pipe* library to evaluate the address offset of these

globals. On variant replication we can surgically update the data at these addresses with the appropriate offset between variants to make these pointers point to the semantically correct global' locations.

Chapter 5

Evaluation

In this chapter, we look at the performance evaluation and security analysis for the Monguard and L-MVX systems. To perform the experiments we used a server class machine running an Intel(R) Xeon(R) Silver 4110 CPU running with a non-boost clock of 2.10GHz with 93.1GB of RAM. We used Debian 9.11 (Stretch) on top of the Linux kernel version 5.2. This processor was chosen as it has the Intel MPK functionality enabled, enabling us to quickly switch read and write permissions for the protected pages.

In the first section we look at the Monguard system's performance and aspects of its security. In the second section this is then repeated for the L-MVX system running on top of Monguard.

5.1 Monguard Evaluation

5.1.1 Performance Evaluation

First, we evaluate the performance overhead of a Monguard-based monitor and a ptrace-based monitor under CPU intensive (SPEC INT2006) and I/O intensive (Nginx, Redis) workloads. SPEC INT2006 consists of several CPU-intensive benchmarks, stressing the system's processor and memory subsystems. In these tests, we use an empty monitor which does not perform any operations so we compare and contrast the frameworks in isolation

without the overhead of running monitor logic. Nginx is a widely used web server in production environment while Redis is a popular key-value store service. In our evaluation, both monitors only intercept the external procedure calls (system calls, libc calls) without any further inspections. Figure 5.1 shows the evaluation results of running these benchmarks. For most CPU intensive workloads, we observe the ptrace monitor and the Monguard monitor perform similarly.

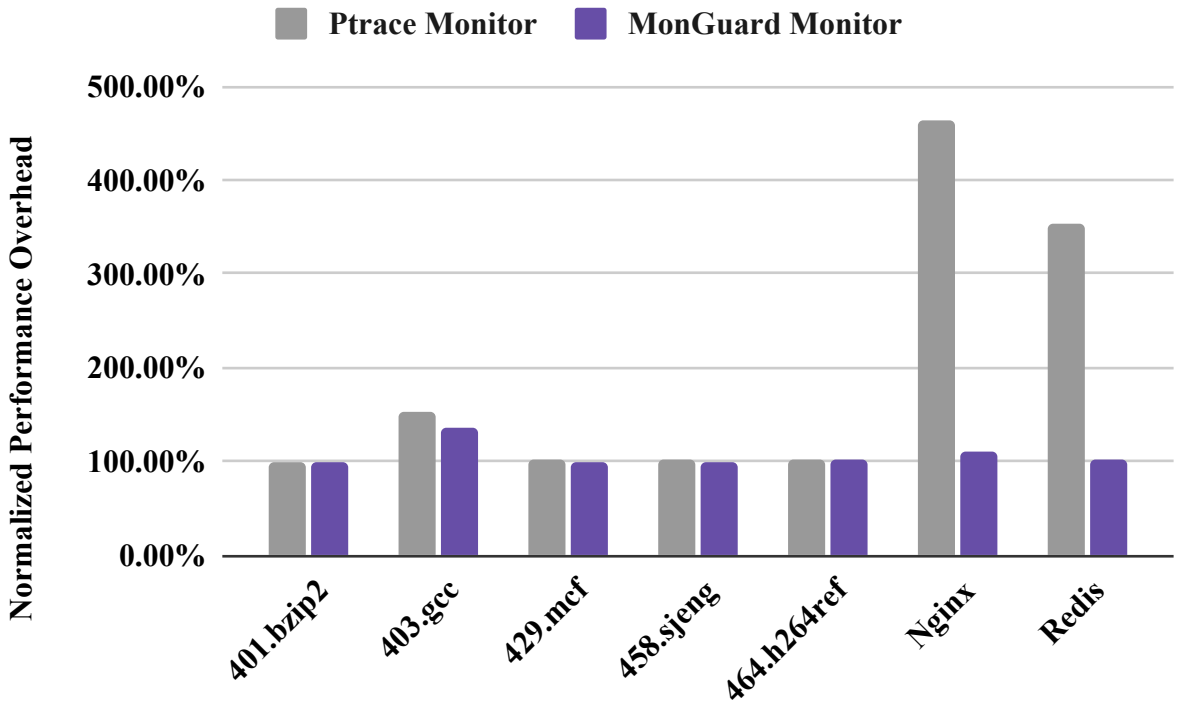

Figure 5.1: Performance overhead of Monguard and a ptrace monitor (Normalized to application running w/o monitors).

However, the ptrace-based monitor brings large performance overheads for I/O intensive work as seen in Nginx at 4.5x and with Redis at 3.5x. This is likely because Nginx and Redis issue more frequent external procedure calls than CPU intensive workload, which amplifies the overhead for the costly out-of-process monitor. More specifically, the two context switches for each system call and the fixed size of the buffer sent to the tracer exacerbates the overhead

when running ptrace on applications like Nginx and Redis.

Another factor to consider in the evaluation discussion when contrasting a ptrace-based monitor with Monguard is the difference in interception techniques. Ptrace-based monitors intercept system calls while our system intercepts library calls. To prove our assumption that the actual number of libc calls being called is not trivial compared to the number of system calls invoked, we profiled the external procedure call rate (number per second). This is shown in figure 5.2, plotted on a log scale. Nginx performs 31,847 system calls per second while 403.gcc performs 9,281 system calls per second. Other applications only issue less than 500 system calls per second on our test machine. The 401.bzip2 benchmark only executes 1.36 syscalls/s with a total of 128 syscalls executed in a single run.

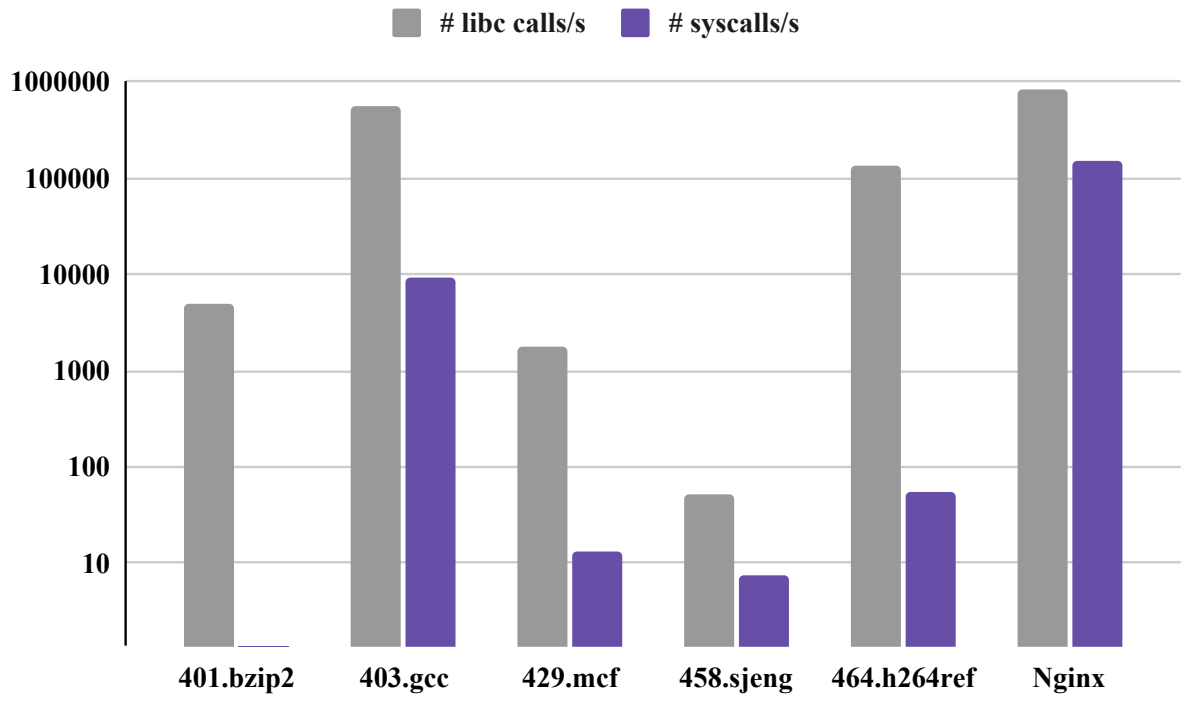

Figure 5.2: Total number of libc calls vs syscalls

On the other hand, we see that the number of libc calls executed is an order of magnitude

higher with Nginx executing 826,931 libc calls/s and 403.gcc coming in second at 570,167 libc calls/s.

In addition, the memory consumption of ptrace-based monitors is shown to be higher than that of in-process monitors due to the additional monitor processes created, while in-process monitors such as Monguard only incur the overhead of the monitor Virtual Memory Area within the process itself.

Finally, to prove the multi-threading capabilities of Monguard, we implemented a simple multi-threading microbenchmark which spawns several threads with the `pthreads` API, and have the threads call a monitored function through the Monguard trampoline, each independently switching into their respective Monguard trampoline stack and unsafe stacks. We then verified that `%rsp` was at different addresses for each thread and there were no data inconsistencies within the library function by incrementing and reading back a value on the stack in the monitored function.

5.1.2 Monguard Security Analysis

Threat Model

From a security perspective, Monguard considers the application itself to contain memory corruption or control-flow hijacking vulnerabilities like buffer overflows allowing it to be exploited by a remote attacker during execution and is categorized as untrusted. However, we also assume the attacker is not a strong attacker and does not have access to the hardware as Intel MPK is incapable of preventing side-channel or microarchitectural attacks like Meltdown [35] and Spectre [28]. At runtime, the attacker can only access the target process remotely through the standard I/O interface, namely socket connections, allowing them to send arbitrary data payloads to the target process. In this case, we also assume the OS ker-

nel is part of the trusted computing base (TCB). Performing isolation using a hypervisor's EPT [37] functionality has the possibility of allowing the kernel to be under attacker control but our approach attempts to bridge the gap between strong monitor isolation and speed; providing lightweight, efficient performance without sacrificing security for the majority of applications.

We also consider the in-process monitor and any protected libraries to be secure and free of such vulnerabilities within itself as triggering a control-flow hijacking attack while in the safe context (where execution is occurring on the Monguard trampoline stack and MPK R/W protections are disabled) nullifies the protections Monguard offers.

While Monguard is incapable of directly detecting or preventing control-flow hijacking attacks the in-process monitor system can be used to provide low latency protection to other monitors capable of detecting and defeating such attacks like our L-MVX MVX system, sandbox and fault isolation system [59, 60], and malware tracers/monitors [74].

MPK Trampoline Safety

The Monguard system utilizes a call-gate approach to ensure the isolation of the in-process monitor. All calls to trusted libraries (including libc and the in-process monitor) from the untrusted application are patched to go through the Monguard trampoline. This means the only legimitate entry points to access these shared libraries is through the call-gate points. The PLT patching techniques discussed in Section 3.3 where the `jmp` instructions in the PLT are patched to jump directly through immediates to the Monguard trampoline means there is no intermediate step where the address of the monitor can be leaked through a pointer to the attacker. To further harden this, the callgate code pages area also marked as *access disabled*, resulting in attackers being unable to read the callgate code memory to locate the

monitor in the address space. We also assume that the 64-bit address space is large enough to hide the in-process monitor and protected libraries.

If the attackers are able to brute-force the location of the monitor, attempting to perform control-flow hijacking attacks like ROP (Return Oriented Programming) to jump into the trampoline code and access the `wrpkru` instruction to clear the *access disabled* bit will result in the attacker touching the barrier variable and triggering an MPK access violation.

Monguard Memory Isolation

Using the call-gate and trampoline the Monguard system leverages MPK to enable the "one-way visibility" of a reference monitor. Monguard logically splits the application address space into an application context and a monitor/library context. Only the monitor/library context is allowed to access the application context, but not vice versa.

To achieve that, Monguard prepares two memory protection keys for the monitor (including the shared libraries) code and data respectively.

The first protection key (PKEY 1) is assigned to all code pages, which includes the code of the monitor, shared libraries and the trampoline call gate. PKEY 1 disallows the read permission of those code pages, making the monitor code execute-only. This can prevent the issue of direct code address leakage, which might be used for dynamical vulnerability discovery and payload generation [52]. Application code can be also optionally assigned with PKEY 1, further preventing attackers from directly reading from the application's code pages. The second protection key (PKEY 2) is associated with the monitor and shared library data pages. The attribute of PKEY 2 will be updated based on the execution context. When the code execution is in the application context, PKEY 2 is set with *access disabled*. However, when the application calls the monitor or library code through the trampoline, the

monitor enables the data access by clearing the *access disabled* bit of PKEY 2 in the PRKU register. Before leaving the monitor context, the monitor sets the *access disabled* bit again, thereby ensuring the monitor's data integrity.

Currently, Monguard isolates both the reference monitor and the shared library data from the application code context, which makes reduces to total attack surface. To achieve that, Monguard intercepts all the library calls and updates PKEY 2 accordingly to allow legal library calls. Monguard does not isolate the stack and heap memory during the application-monitor context switch. In Monguard's design, we assume the monitor does not call the application code. Therefore, the application code will never have a chance to manipulate the monitor's stack. Heap is another potential target for attacker. In current monitor implementation, we did not use any dynamic memory. However, we do provide a hooked Monguard `mmap` implementation to associate 4k pages with PKEYs. A more comprehensive solution might be to embed a simple `malloc()/free()` implementation on top of the protected `mmap`'ed memory.

Monguard Trampoline Stack Isolation

The use of separate stack when executing in the protected monitor or library context prevents us the Monguard system from leaking addresses or data from the protected context on the stack. As has been demonstrated with non-control-data attacks, the primary vector of these attacks are local stack variables or register variables spilled from the caller and saved on the stack. When entering and leaving the protected contexts, we swap stack execution between Monguard trampoline stacks and the regular application stack, henceforth referred to as the unsafe stack. This further prevents the leakage of sensitive data or pointers internal to the security monitor and guarded libraries, namely configuration data, session data, and other data which is part of the decision-making process.

Stack-based buffer overflows within the untrusted application cannot access the Monguard trampoline stack as it is further isolated by associating the Monguard trampoline stacks pages to PKEY 2, disallowing illegitimate data accesses not through the Monguard trampoline.

5.2 L-MVX Evaluation

5.2.1 L-MVX Performance Evaluation

Next, we evaluate the performance of the L-MVX system running on top of the Monguard in-process monitor. L-MVX is compiled as a shared library and pre-loaded when running the target application.

Currently, the MVX monitor simulates 35 libc calls for the follower variant execution. For example, the file descriptor related functions such as `fopen`, `fdopen`, `close`; the networking and event poll related functions such as `epoll_create`, `sendfile`, `writev`, etc.

We first chose to evaluate L-MVX's performance on the Linux/Unix release of the BYTEmark benchmark suite, also known as Nbench [11]. These benchmarks show how L-MVX performs when run against benchmarks heavily utilizing the system's CPU, FPU and memory system. Figure 5.3 shows the normalized results from our experiments. Similar to the results of Monguard, executing L-MVX on programs which are CPU-bound instead of I/O-bound gives us very promising performance numbers, showing minimal performance overhead, with the highest overad seen in the Neural Net benchmark due to its relatively high I/O usage compared to the other tests (from reading the model file.).

But how does L-MVX perform on I/O heavy workloads? We measured the performance of running two different webservers. The first test result is from two Nginx variants, choosing to go with the worst-case performance of L-MVX, protecting the entire program call graph

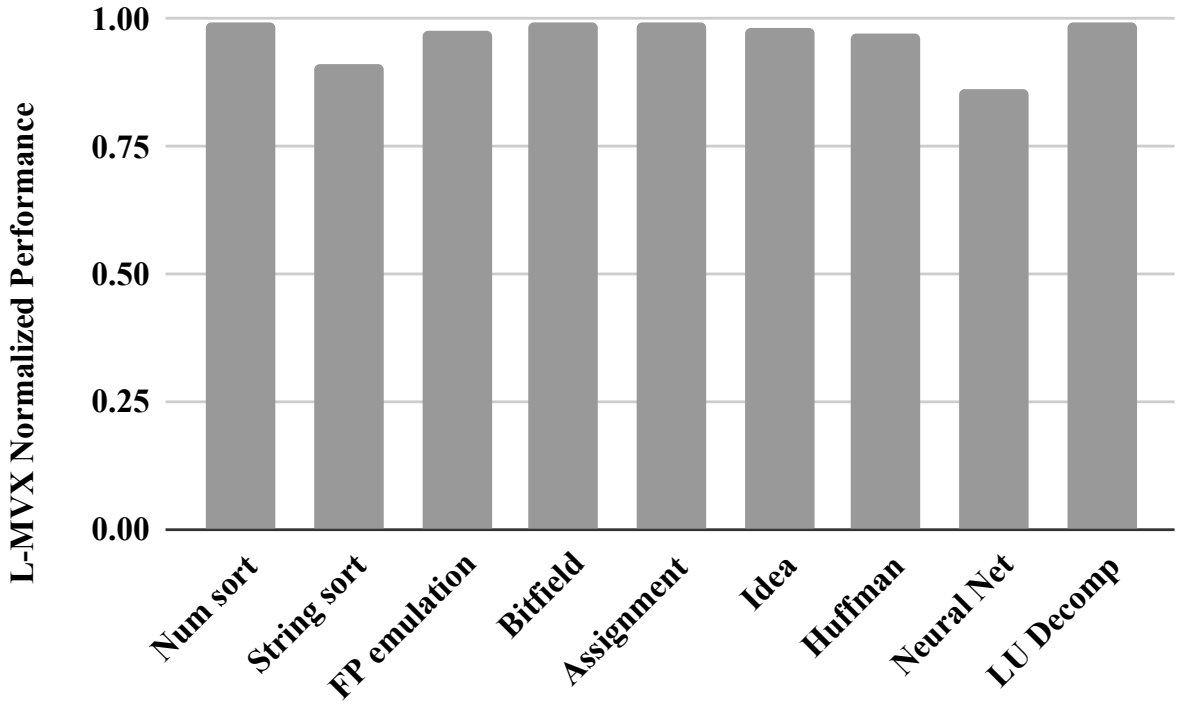

Figure 5.3: nbench execution guarded with L-MVX

as a baseline. As we reduce the size of the protected call graph to only encompass the functions in our taint analysis we expect performance of L-MVX to increase accordingly, with the exception of functions directly in the control loop of the program as that would repetitively incur the overhead from process duplication and pointer updates. This is reflected in the number of procedure calls to the PLT (primarily libc library) that would need to be duplicated and checked across the protected lifetime of the program when benchmarked with 100k http requests shown in Figure 5.4. The purple triangles denote the functions deemed tainted through our taint analysis performed in Section 4.5, showing significantly fewer PLT calls that need to be duplicated and checked over protecting the main function.

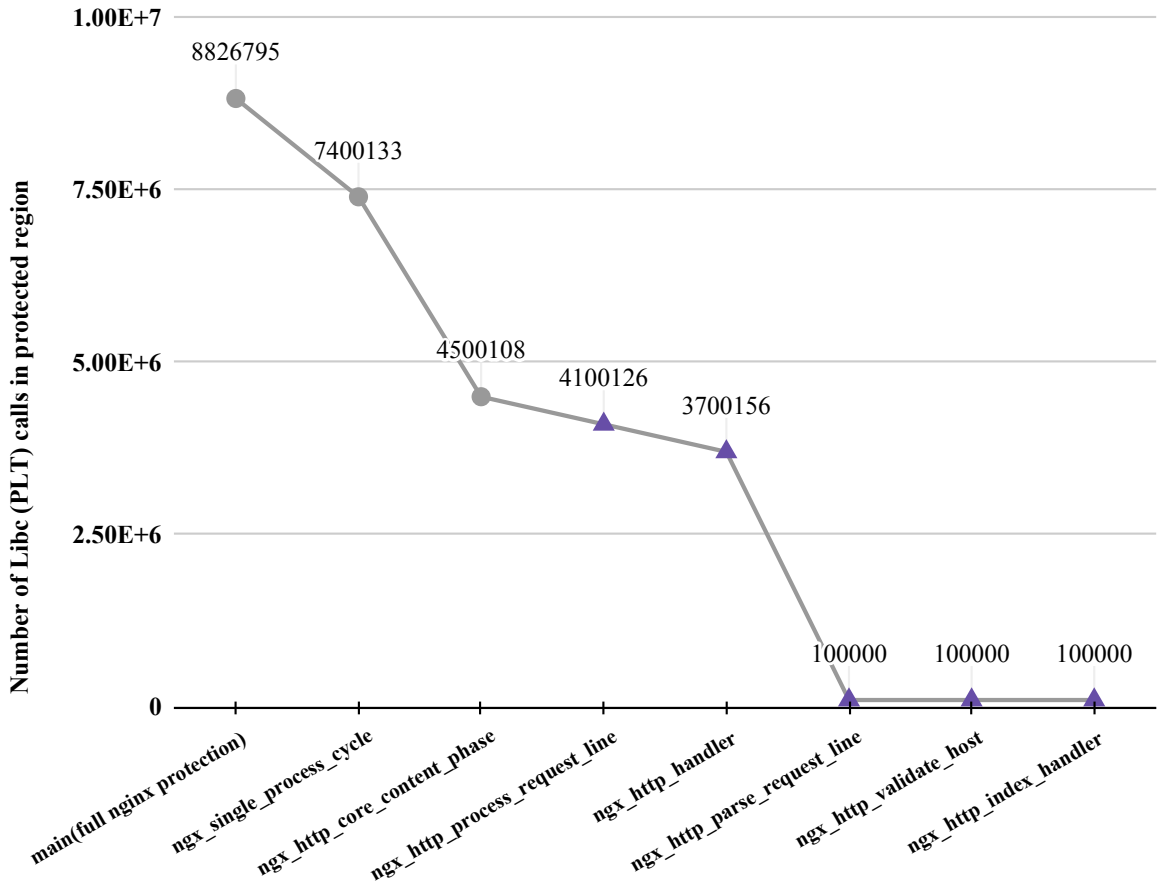

Figure 5.4: Number of Libc (PLT) calls within protected region in Nginx given 100k requests

To run the I/O heavy evaluation on our target webservers protected with L-MVX, we used the `ApacheBench` benchmark as the workload generator and tested on the loopback network interface (0.1ms network latency) as mentioned in a state-of-the-art MVX system (ReMon [63]). The page size that we were serving from the webservers was 4KB in length. With L-MVX running on Monguard we achieve a *2.7x* overhead for Nginx workload, which outperforms the *3x* overhead of ReMon [63] as shown in Figure 5.5. We also tested the lighttpd webserver, where the performance of L-MVX fully protecting the entire call-graph of lighttpd is less than that of the ReMon system with an overhead of *2.3x* as opposed to ReMon's overhead of *1.5x*. This overhead stems from the sheer number of libc calls lighttpd

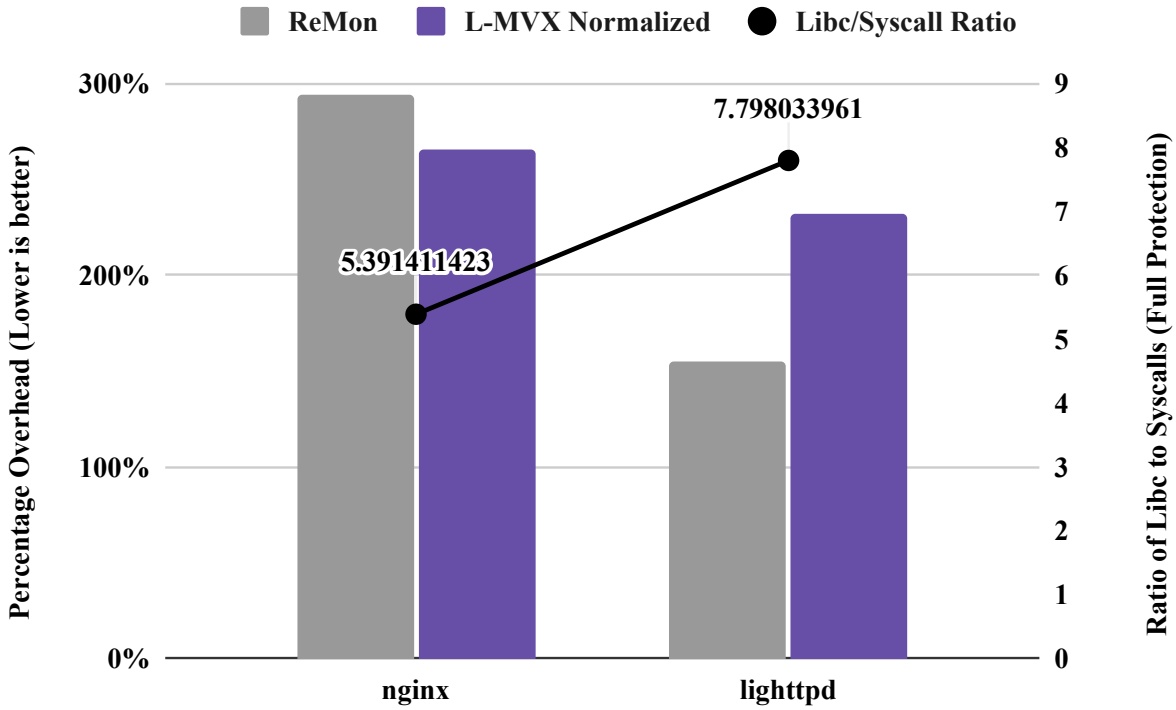

Figure 5.5: Nginx and Lighttpd performance guarded with L-MVX

issues (that will eventually have to be intercepted and checked/simulated) over the system calls it invokes, as seen by the higher ratio of libc to system calls of *7.8* compared to *5.4* of nginx.

The Problem With Control Loops

When running L-MVX and protecting functions within control loops, we ran into a critical issue - the overheads involved in the setup of the second variant, especially when attempting to move the application's virtual memory areas mid-execution while maintaining semantic correctness dominate the savings we receive from not having to perform MVX lockstep execution on untainted portions of the application. A breakdown of these overheads is shown in Table 5.1. We observe that the latency from just the process duplication itself is trivial

Source	Latency (μs)
Process Duplication Overhead (copy+move)	14.7
Data Pointer Scan Overhead	320.8
Heap Pointer Scan Overhead	13162.4

Table 5.1: `lmvx_start()` overheads on `lighttpd`

compared to the overheads seen when scanning the heap for code or data pointers pointing to the original `.data`, `.bss` or `.text` locations when performing the start-up function `lmvx_start()` on the `lighttpd` webserver's main loop function.

5.2.2 L-MVX Security Analysis

Similar to existing diversification systems and Monguard, we assume the attackers have remote access to the target process with a known interface (e.g., connection sockets). The target application is again treated as untrusted and may potentially contain memory corruption vulnerabilities.

Finally we take a look at the security guarantees that the L-MVX system provides. These security guarantees of L-MVX are not that different from those found in other MVX systems [45, 68] as we are relying on an variation between two variants to illicit a different response (whether it be syscall call sequence or libc call sequence) when input with the same attack payload. The variation between Variant 1 and Variant 2 in L-MVX is provided by the non-overlapping address spaces of each variant. This results in control-flow-hijacking attacks such as ROP or return-to-libc failing to execute ROP chains containing gadgets found in the program's address space consistently between variants as the locations of these gadgets will differ, causing memory access errors in Variant 2 when attempting to jump to original gadget locations from Variant 1. If an attacker attempts to modify the program stack data

through non-control-data attack via a buffer overflow vulnerability and cause a divergence in program execution the MVX engine will throw a fault and alarm the monitor system due to differing libc function call sequences between the two variants. This can also be detected in some cases when comparing the input arguments to libc calls.

Nginx CVE 2013-2028

To evaluate the security of L-MVX we sought to reproduce a memory corruption vulnerability on one of our guarded applications. In particular, we chose to reproduce a stack-based buffer overflow vulnerability occurring on Nginx, allowing us to perform an out-of-bounds write, overwriting control data of the application and gain control of the control flow of the webserver. The experiment was performed on the Nginx webserver version 1.3.9 which contained this vulnerability.

The bug in nginx allowed a remote attacker to specify the size of a chunked http request when issuing a http request with a header "Transfer-Encoding:chunked". This value is defined as an unsigned number, which can later be misrepresented as a negative integer when cast to a signed type, which is later re-casted from a negative signed value to an unsigned and used as the size of data to read. As shown in Listing 5.1, the target function being exploited, `ngx_http_read_discarded_request_body` contains a buffer that will receive data from the attacker via the `recv` libc library call, with the goal being to trick `recv` to perform a buffer out-of-bounds write into that buffer. `r->headers_in.content_length_n` is indirectly under attacker control, ultimately allowing it to be a negative number which when casted to `size_t`, becomes a large positive number. More details can be found in the writeup by vnsecurity [65].

This buffer overflow allows us to begin a Return-Oriented-Programming attack on nginx. In

our exploit, we performed a simple ROP chain using gadgets found in the program address space utilizing the Ropper [48] and ROPGadget [26] tools. Since we are not attempting to prove the turing-completeness of ROP chains, the actual logic being performed is of less importance than its observability. Our ROP chain consists of 3 gadgets and 3 values, loading a pointer to a string found in the application to `%rdi`, popping an integer from the stack to `%rsi`, and jumping to the location of the `mkdir` libc call to create a directory.

```
1  static ngx_int_t
2  ngx_http_read_discarded_request_body(ngx_http_request_t *r)
3  {
4      size_t     size;
5      ssize_t    n;
6      ngx_int_t  rc;
7      ngx_buf_t  b;
8      u_char     buffer[NGX_HTTP_DISCARD_BUFFER_SIZE];
9
10     ...
11
12     size = (size_t)ngx_min(r->headers_in.content_length_n,
       NGX_HTTP_DISCARD_BUFFER_SIZE);
13     n = r->connection->recv(r->connection, buffer, size);
```

Listing 5.1: Nginx vulnerable function

Running the exploit on nginx protected by L-MVX we observe that Variant 2 throws a fault when the program counter tries to jump to gadget locations that were present in Variant 1's address space but were otherwise unmapped in Variant 2, thereby detecting and breaking the attack. In addition, it should be noted that `recv` is one of the I/O libc calls simulated on Variant 2. This opens up the possibility of an in-process monitor performing extra bounds checks on sensitive calls in the future to prevent such attacks from occurring.

Chapter 6

Conclusions

In this thesis we highlighted the drawbacks of current monitor isolation techniques, namely the latency cost of performing context switches to the kernel and subsequently to another process address space and the intrusiveness of introducing some of these systems, requiring modifications to the kernel or heavy compiler instrumentation. We then explored improvements which could be made in these areas by introducing Monguard, an in-process monitor protection system using Intel MPK technology to implement execute-only memory in tandem with runtime binary patching and a trampoline which creates a strong isolation between protected contexts and application contexts. Monguard demonstrates promising results with almost no overhead when run on CPU intensive workloads (SPEC INT2006) and significantly better overheads compared to `ptrace` monitors on I/O heavy workloads (nginx, redis).

Following that, we surveyed existing program monitors and Multi Variant Execution (MVX) systems and discussed their weaknesses. Many such systems also utilized the `ptrace` interface, leading to high context switching overheads when switching to the tracer process. They also suffered from a high number of false positives due to isolation and non-determinism between variants. To address these concerns we introduced L-MVX, a flexible lightweight in-process Multi Variant Execution (MVX) monitor running on the Monguard protection framework. L-MVX is capable of protecting flexible regions (sub-trees) in the program as opposed the entire program call-graph, allowing us to focus on protecting only the code which touches tainted data that originate from an attacker (eg. network sockets). It also

intercepts libc calls instead of syscalls and demonstrated close-to-vanilla performance on the Nbench benchmark suite, and better performance compared to state-of-the-art (ReMon)systems on the Nginx webserver when protecting the entire call graph. While lighttpd suffered from lower performance when compared to ReMon, this was primarily due to a high libc to syscall ratio and the fact that the entire call graph was protected. A drawback of L-MVX was the poor performance while duplicating functions within a tight control loop due to heavy overheads of runtime process address space shifting and resultant updates required to maintain semantics. To solve this issue we looked to static analysis and compiler instrumentation techniques.

In summary, our contributions in this thesis are as follows. We introduced Monguard, a system in which a high-performance in-process monitor is efficiently isolated from the rest of the application leveraging Intel Memory Protection Key (MPK) technology, Procedure Linkage Table (PLT) call trampolines, and safe execution stack contexts. Next we unveiled L-MVX, a practical use for the Monguard system in the form of a lightweight in-process Multi Version Execution (MVX) monitor, allowing us to perform selective MVX protection at a PLT library call granularity on vulnerable sections of the program, further reducing the overhead of cross-variant monitor synchronization. To solve problems that arose due to the tight control loop, we discussed static analysis methods using LLVM compiler framework middle-end passes to further increase the flexibility and efficiency of L-MVX, allowing runtime memory diversification without breaking program semantics.

One of the core tenets of software design is that there are *tradeoffs* in designing all systems and this too is true in state of the art of memory isolation, protection domains, and MVX systems today. We believe that in some cases, the tradeoff of stronger isolation boundaries for speed by enforcing these boundaries with the help of hardware features like Intel MPK can be the practical choice for memory isolation techniques. Shedding unnecessary protection

from MVX systems in areas that are not vulnerable to attack in exchange for reduced false-positive detection rates and potentially better performance also proves to be a more flexible and robust approach. In the next section we discuss the limitations of the current systems and explore ways in which we can further improve upon them from security and performance standpoints.

6.1 Limitations and Future Work

While it is an interesting technology, the usage of Intel MPK in in-process monitors comes with its limitations and caveats. Firstly, there is a limited set of protection keys that may not suit all monitor scenarios. This limitation can be removed by virtualizing the keys [42], however this comes at the cost of lower performance, another tradeoff. A second limitation stems from the fact that the PKRU switch operation is, for performance reasons, an unprivileged instruction. Combined with the fact that MPK does not check memory accesses on instruction fetches, it raises concerns about PKRU manipulation by untrusted code, either directly or indirectly through techniques such as Return-Oriented Programming (ROP). Thus, isolation schemes must be complemented with static code analysis [61] to validate each update of the PKRU register. Sequences of bytes forming PKRU manipulating instructions may also appear due to the variable size of the x86-64 instruction set, in a similar manner as ROP gadgets. Solution have been proposed including binary rewriting techniques [61] as well as traps based on hardware watchpoints [21] to address that issue.

Finally, simply updating the PKRU upon security domain switch does not prevent the leakage of registers content between domains. This can be exploited to mount an attack. Although the solution is to save, scrub and restore registers values, this may impact the latency of security domain switch operations. Similarly this problem can also be observed with

`syscall` instructions, namely in the pure monitor scenario without an MVX engine, there is nothing stopping an attacker from jumping to ROP gadgets within the untrusted application code, potentially finding and utilizing these `syscall` instructions through unaligned accesses thereby allowing the attackers to affect the system without tripping any MPK faults.

We currently do not support multi-threading with L-MVX. However, it might be possible to hook the `pthread_create` calls with a `LD_PRELOAD`ed version which spawns an alternate variant for that thread. This would also require setting up a separate shared memory IPC communication area for the thread and its variant and associating the pages with an MPK key. Next, while L-MVX is capable of protecting specific regions in the program, we are still blocked by the poor performance of protected regions within a tight control-loop. It remains the case that during every execution of the protected region we have to scan the entire address space for pointers as seen in Section 4.6. We have made progress towards solving this bottleneck in Section 4.7 but there still remains work to be done to solve the other two cases, specifically global data pointers on the heap and code pointers in both data and the heap at process duplication time.

6.1.1 Towards Finding Global Data Pointers on the Heap

To solve the problem of global data pointers on the heap, we have an untested idea and implementation which also leverages static analysis and function call replacement using a LLVM IR middle-end pass. This heavily leverages the **Static Value-Flow analysis framework** created by Sui et al. [53, 54, 55]. We propose the algorithm shown from a high level in Listing 6.1. First, `MVXAA::initialize()` builds the SVF module from the merged LLVM bitcode module, runs the SVF implementation of Andersen's Wave Propagation analysis, and generates the Static Value-Flow Graph. The `malloc` callsite nodes (also known as source nodes in source-sink analysis) are also collected and stored in the `sourceNodes` set. Subsequently, `MVXAA::collectProtectedFunctions()` walks through the callgraph generated by SVF and collects all direct and indirect calls made by the protected function and stores them in the set `protectedFuncNodes`.

```
Set<SVF::VFGNode* > globalAddressUseNodes
Set<SVF::VFGNode* > sourceNodes
Set<SVF::SVFFunction* > protectedFuncNodes
Map<SVF::VFGNode*, Set<int>> sourceToOffsetsMap

MVXAA::driver() {
    initialize() // Initialize SVF and get PTACallgraph, SVFG
    collectProtectedFunctions()
    findGlobalAddressUses()
    collectUsesFlowingFromSourceNodes()
    replaceMallocs()
}
```

Listing 6.1: Heap global address detection algorithm pseudocode

As seen in Listing 6.2, `MVXAA::findGlobalAddressUses()` next looks at all Global Nodes in the SVFG and iterates over them, checking if the globals have their addresses stored into a pointer. In the SVFG this is represented by a Global node flowing to a Store node. Next, the algorithm performs `Breadth-first Graph Traversal` using a FIFO worklist on the nodes downstream of the Store node to locate any uses of the global's address and stores them into the `globalAddressUseNodes` set.

```
MVXAA::findGlobalAddressUses() {
    foreach globalNode in SVFG:
        // Does the global node's address value flow to a store node?
        if node has immediate child (Store Node):
            // Traverse down SVFG from that point:
            foreach nextNode in breadthFirstTraversal(node)):
                if useNode is a use of globalNode:
                    // Insert into globalAddressUse set:
                    globalAddressUseNodes.insert(useNode)
}
```

Listing 6.2: `MVXAA::findGlobalAddressUses` pseudocode

`MVXAA::collectUsesFlowingFromSourceNodes()` (Listing 6.3) iterates over the previously collected source nodes and performs a **Depth-First Traversal** on the graph, checking to see if any of its child nodes also belong in the `globalAddressUseNodes` set collected in the previous step, while also adding or subtracting any constant GEP offsets encountered along a particular path. If there are child nodes of the source in the `globalAddressUseNodes` set, it means a global address was stored into heap memory. The algorithm then stores this source node and corresponding offset along the path into the `sourceToOffsetsMap`.

```
MVXAA::collectUsesFlowingFromSourceNodes {
    foreach sourceNode in sourceNodes:
        // Traverse down SVFG from that point:
        foreach nextNode in depthFirstTraversal(node)):
            // If node is in global address use set and if its
            // containing function is in the protected CG:
            if nextNode is in globalAddressUseNodes AND
                nextNode->parent->parent is in protectedFuncNodes :
                sourceToOffsetMap[nextNode].insert(offsetAtDFTLevel)
            // Else collect GEP offset if it's a GEP instruction:
            else if nextNode is a GEP node:
                offsetAtDFTLevel += nextNode.offset
}
```

Listing 6.3: `MVXAA::collectUsesFlowingFromSourceNodes` pseudocode

We then proceed to replace the corresponding `malloc` call instruction with a call to a custom `malloc` function shown in Figure 6.1, also passing in the offsets at which the `load` instruction or instructions were loading from as additional parameters. The LLVM bitcode is then compiled into an ELF binary and run with L-MVX.

Malloc call instruction example and replacement:

```
%call = call noalias i8* @malloc(i64 %size);
%call = call noalias i8* @new_malloc(i64 %size, i64 %offset0, ...);
```

Figure 6.1: L-MVX Duplicated Execution only for input handling code

Finally, at runtime, this `new_malloc` function is intercepted via `LD_PRELOAD` (Listing 6.4). We pass in only the size argument to the real `malloc` and call it, getting the runtime-allocated address from `malloc`. We can then insert the returned address added with the offset or offsets which were passed in as constants from the prior static analysis pass to a

new set called `updateSet`. Similar to the globals offset update in Section 4.7.1 we can then surgically update only those addresses with the offset of the moved process address space on variant replication without having to iteratively scan the heap for these pointers.

```
void* new_malloc(size_t n, uint64_t offset0, uint64_t offset1, ...) {
    /* Call real malloc and save the returned address */
    void* malloc_address = malloc(n);

    /* Store address + offset to updateSet */
    foreach offset:
        updateSet.insert(malloc_address+offset);

    return malloc_address;
}
```

Listing 6.4: `LD_PRELOAD new_malloc` interception

Bibliography

[1] *System V Application Binary Interface AMD64 Architecture Processor Supplement*, 0.99.7 edition, November 2014. https://www.uclibc.org/docs/psABI-x86_64.pdf, Online, accessed 11/26/2020.

[2] One Aleph. Smashing the stack for fun and profit. *http://www.shmoo.com/phrack/Phrack49/p49-14*, 1996.

[3] aslrlinux. Linux Kernel Address Space Layout Randomization. http://lwn.net/Articles/569635/.

[4] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You Can Run but You Can'T Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, 2014.

[5] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

[6] Emery D Berger and Benjamin G Zorn. Diehard: probabilistic memory safety for unsafe languages. *Acm sigplan notices*, 41(6):158–168, 2006.

[7] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. Hacking Blind. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 227–242. IEEE, 2014.

[8] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. Diversified process replicæ for defeating memory error exploits. In *2007 IEEE International Performance, Computing, and Communications Conference*, pages 434–441. IEEE, 2007.

[9] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, October 2008.

[10] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining light on shadow stacks. *arXiv preprint arXiv:1811.03165*, 2019.

[11] BYTEmark benchmark. Linux/Unix nbench. http://www.math.utah.edu/~mayer/linux/bmark.html, Accessed: 2020-11-14.

[12] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, 2006.

[13] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of 2005 USENIX Security Symposium*, August 2005.

[14] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *USENIX Security Symposium*, pages 105–120, 2006.

[15] Data execution prevent. x86 NX support. http://lwn.net/Articles/87814/.

[16] David Mulnix. Intel® Xeon® Processor Scalable Family Technical Overview. https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview, Accessed: 2020-02-14.

[17] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and et al. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, page 475–488, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450332132. doi: 10.1145/2663716.2663755. URL https://doi.org/10.1145/2663716.2663755.

[18] Rich Felker. musl libc., 2020. https://musl.libc.org/, Online, accessed 11/26/2020.

[19] Linux Foundation. *ld.so(8) Linux User's Manual*, 2.9.3 edition, August 2020.

[20] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. {IMIX}: In-process memory isolation extension. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 83–97, 2018.

[21] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 489–504, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8.

[22] Petr Hosek and Cristian Cadar. Varan the unbelievable: An efficient n-version execution framework. *ACM SIGARCH Computer Architecture News*, 43(1):339–353, 2015.

[23] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang.

Automatic generation of data-oriented exploits. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 177–192, 2015.

[24] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. IEEE, 2016.

[25] *Software Guard Extensions Programming Reference*. Intel, Sep 2013.

[26] JonathanSalwan. Ropgadget github webpage, 2020. https://github.com/JonathanSalwan/ROPgadget, Online, accessed 11/26/2020.

[27] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 121–132, 2012.

[28] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[29] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and Efficient Multi-Variant Execution using Hardware-Assisted Process Virtualization. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 431–442. IEEE, 2016.

[30] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos.

No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 437–452. ACM, 2017.

[31] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4.

[32] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated Software Diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, 2014.

[33] Hojoon Lee, Chihyun Song, and Brent Byunghoon Kang. Lord of the x86 rings: A portable user mode privilege separation architecture on x86. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1441–1454, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243748. URL https://doi.org/10.1145/3243734.3243748.

[34] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann, San Francisco, CA, 1999.

[35] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018.

[36] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-weight contexts: An OS abstraction for safety and performance. In *12th USENIX Symposium on Operating Systems Design and Implemen-*

tation (OSDI 16), pages 49–64, Savannah, GA, November 2016. USENIX Association. ISBN 978-1-931971-33-1.

[37] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1607–1619, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338325. doi: 10.1145/2810103.2813690. URL https://doi.org/10.1145/2810103.2813690.

[38] llvm. The LLVM Compiler Infrastructure. http://llvm.org/.

[39] Kangjie Lu. How to make aslr win the clone wars: Runtime re-randomization. 2016.

[40] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System v application binary interface. 2013.

[41] MITRE 2020. 2020 cwe top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html, 2020.

[42] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8.

[43] Radare Org. R2pipe github webpage, 2020. https://github.com/radareorg/radare2-r2pipe, Online, accessed 11/18/2020.

[44] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. http://cseweb.ucsd.edu/ hovav/dist/rop.pdf, 2009.

[45] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 33–46. ACM, 2009.

[46] Salamat, Babak and Gal, Andreas and Franz, Michael. Reverse stack execution in a multi-variant execution environment. In *Workshop on Compiler and Architectural Techniques for Application Reliability and Security*, pages 1–7, 2008.

[47] *System V Application Binary Interface Intel386 Architecture Processor Supplement*. The Santa Cruz Operation, Inc, fourth edition edition, March 1997. http://www.sco.com/developers/devspecs/abi386-4.pdf, Online, accessed 11/26/2020.

[48] Sascha Schirra. Ropper github webpage, 2020. https://github.com/sashs/Ropper, Online, accessed 11/26/2020.

[49] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-Into-Libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, October 2007.

[50] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, page 477–487, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605588940. doi: 10.1145/1653662.1653720. URL https://doi.org/10.1145/1653662.1653720.

[51] Ajey Singh and Dr Maneesh Shrivastava. Overview of attacks on cloud computing. *International Journal of Engineering and Innovative Technology (IJEIT)*, 1(4), 2012.

[52] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher

Liebchen, and Ahmad-Reza Sadeghi. Just-in-time Code Reuse: On the Effectiveness of Fine-grained Address Space Layout Randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[53] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.

[54] Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 40(2):107–122, 2014.

[55] Sui, Yulei and Xue, Jingling. Svf github webpage, 2021. https://github.com/SVF-tools/SVF, Online, accessed 19/01/2021.

[56] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal War in Memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.

[57] Trail Of Bits. Mcsema github webpage, 2020. https://github.com/lifting-bits/mcsema, Online, accessed 11/10/2020.

[58] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In *International Workshop on Recent Advances in Intrusion Detection*, pages 121–141. Springer, 2011.

[59] Chia-Che Tsai. *A Library Operating System for Compatibility*. PhD thesis, State University of New York at Stony Brook, 2017.

[60] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. Cooperation and security isolation of library oses for multi-process applications.

In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[61] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. Erim: Secure, efficient in-process isolation with protection keys (mpk). In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 1221–1238, USA, 2019. USENIX Association. ISBN 9781939133069.

[62] Stijn Volckaert, Bjorn De Sutter, Tim De Baets, and Koen De Bosschere. Ghumvee: Efficient, effective, and flexible replication. In *International Symposium on Foundations and Practice of Security*, pages 261–277. Springer, 2012.

[63] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. Secure and efficient application monitoring and replication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 167–179, Denver, CO, June 2016. USENIX Association. ISBN 978-1-931971-30-0.

[64] Alexios Voulimeneas, Dokyung Song, Fabian Parzefall, Yeoul Na, Per Larsen, Michael Franz, and Stijn Volckaert. Dmon: A distributed heterogeneous n-variant system. *arXiv preprint arXiv:1903.03643*, 2019.

[65] w00d. Analysis of nginx 1.3.9/1.4.0 stack buffer overflow and x64 exploitation (CVE-2013-2028)). https://www.vnsecurity.net/research/2013/05/21/analysis-of-nginx-cve-2013-2028.html.

[66] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-based Fault Isolation. In *Proceedings of the 14th ACM Symposium On Operating System Principles*, December 1993.

[67] Xiaoguang Wang, Yue Chen, Zhi Wang, Yong Qi, and Yajin Zhou. Secpod: A framework for virtualization-based security systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 347–360, Santa Clara, CA, July 2015. USENIX Association. ISBN 978-1-931971-225. URL https://www.usenix.org/conference/atc15/technical-session/presentation/wang-xiaoguang.

[68] Xiaoguang Wang, SengMing Yeoh, Robert Lyerly, Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. A framework for software diversification with {ISA} heterogeneity. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*, pages 427–442, 2020.

[69] Xiaoguang Wang, SengMing Yeoh, Pierre Olivier, and Binoy Ravindran. Secure and efficient in-process monitor (and library) protection with intel mpk. In *Proceedings of the 13th European workshop on Systems Security*, pages 7–12, 2020.

[70] Wikipedia. Buddy Memory Allocation. http://en.wikipedia.org/wiki/Buddy_memory_allocation, .

[71] Wikipedia. Ptrace. http://en.wikipedia.org/wiki/Ptrace, .

[72] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *OSDI*, pages 367–382, 2016.

[73] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Orm, Shiki Okasaka, Neha Narula, Nicholas Fullagar, and Google Inc. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, May 2009.

[74] Min Zheng, Mingshen Sun, and John CS Lui. Droidtrace: A ptrace based android dynamic analysis system with forward execution capability. In *2014 international wireless communications and mobile computing conference (IWCMC)*, pages 128–133. IEEE, 2014.

[75] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 558–569, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450329576. doi: 10.1145/2660267.2660344. URL https://doi.org/10.1145/2660267.2660344.

[76] Sebastian Österlund, Koen Koning, Pierre Olivier, Antonio Barbalace, Herbert Bos, and Cristiano Giuffrida. kMVX: Detecting Kernel Information Leaks with Multi-variant Execution. In *ASPLOS*, April 2019.