Vrije Universiteit Amsterdam

Master Thesis

Detecting information leaks using kernel-level multi-variant execution

Author:
Sebastian Österlund

Supervisors:
Prof. Dr. Herbert Bos
Dr. Cristiano Giuffrida
Koen Koning

A thesis submitted in fulfilment of the requirements for the degree of Master of Parallel and Distributed Computer Systems

in the

Faculty of Science
Department of Computer Science

August 2017

VRIJE UNIVERSITEIT AMSTERDAM

Abstract

Faculty of Sciences
Department of Computer Science

Master of Parallel and Distributed Computer Systems

Detecting information leaks using kernel-level multi-variant execution

by Sebastian ÖSTERLUND

Kernel information leak vulnerabilities are a major security threat to production systems. By exploiting these commonly found bugs, attackers can leak confidential information, such as cryptographic keys or kernel base pointers, helping them defeat existing defenses, such as Kernel Address-Space Layout Randomization, in a later stage of an attack.

In this paper we present kMVX, a comprehensive kernel-level multi-variant execution (MVX) system designed to prevent kernel information leaks. kMVX makes use of a multi-kernel design, where multiple kernels variants are run in parallel within the a single machine. We demonstrate a implementation of this design on top of Popcorn Linux, a multi-kernel fork of Linux. In this implementation we utilize the message-passing framework, provided by Popcorn Linux, to compare the state of the kernels at certain synchronization points.

We show that a carefully selected variant generation strategy for the kernels can be used to effectively detect a majority of kernel information leaks. We show that, not only is kMVX a comprehensive defense against multiple real-world kernel information leak vulnerabilities, but that our design can be non-intrusive enough to use on production systems. By utilizing our policy manager the performance can be tweaked further to target specific use-cases.

Finally, we utilize this system to effectively defend the kernel against hardware-based side-channel timing attacks, commonly used to leak confidential information.

Acknowledgements

I would like to thank Antonio Barbalace and other people of the Popcorn Linux team for providing their code base for their implementation kMVX. I would like to thank Koen Koning for his implementation of the type-safe kernel memory allocator and stack variation techniques, as well as for his input on this thesis. I would also like to thank my supervisors, Herbert Bos and Cristiano Giuffrida for their support and insights. Finally, I would like to thank my fellow Teaching Assistants at the VU for their moral support in times of despair. I would especially like to thank Joost Heitbrink for proof-reading this thesis.

Contents

Abstract **i**

Acknowledgements **ii**

List of Figures **v**

List of Tables **vi**

1 Background **3**
- 1.1 Kernel vulnerabilities . . . 3
  - 1.1.1 Types of exploits . . . 3
  - 1.1.2 Information leakage . . . 4
- 1.2 Multi-Variant Execution . . . 5
- 1.3 Side-channel attacks . . . 8
- 1.4 Multikernels . . . 9

2 Design **11**
- 2.1 Threat model . . . 11
- 2.2 Kernel vs. virtualization vs. hardware MVX . . . 12
- 2.3 Kernel-level MVX . . . 13
- 2.4 Monitor . . . 14
  - 2.4.1 Sources and sinks . . . 15
- 2.5 Severity level manager . . . 16
  - 2.5.1 Abnormal behavior detection . . . 17
- 2.6 Variant Generation . . . 17

3 Implementation **19**
- 3.1 Utilizing Popcorn Linux for MVX . . . 20
  - 3.1.1 Multi-kernel design . . . 20
  - 3.1.2 FT-Linux . . . 20
- 3.2 Monitor . . . 21
  - 3.2.1 Copy-to-user synchronization . . . 22
  - 3.2.2 Copy-from-user synchronization . . . 24
  - 3.2.3 System call modifications . . . 25
- 3.3 Severity level manager . . . 26

3.3.1 Performance vs. security goals . . . 26
3.3.2 Activity Detectors . . . 27
3.3.3 Side-channel detection . . . 27
3.4 Variant generation . . . 28
3.4.1 Non-overlapping virtual address space . . . 28
3.4.2 Variation in memory allocators . . . 29
3.4.3 Stack format . . . 30

4 Results **32**
4.1 Threat model . . . 32
4.2 Performance evaluation . . . 32
4.2.1 Experimental setup . . . 32
4.2.2 Results . . . 35
4.3 Security Analysis . . . 39

5 Discussion **43**
5.1 Why kMVX? . . . 43
5.2 Drawbacks . . . 43
5.2.1 Need for more kernel variants . . . 44
5.2.2 Single point of failure . . . 44
5.2.3 Replay vs. multi-variant . . . 44
5.3 Future extensions . . . 45
5.3.1 Network-performance improvement . . . 45
5.3.2 vDSO . . . 45
5.3.3 Remaining sources of divergence . . . 45
5.3.4 File system . . . 46
5.3.5 Inter-kernel communication . . . 46
5.4 Conclusion . . . 47

Bibliography **48**

List of Figures

2.1 kMVX design overview. The master kernel communicates with the outside world, while the replica's verify that their state is the same as the master's. 14

3.1 `copy_to_user` synchronization in kMVX. 23
3.2 `copy_from_user` synchronization in kMVX. 25
3.3 kMVX multikernel design. The kernels communicate through message passing. 27
3.4 Partitioned virtual address space in kMVX. 29

4.1 Macro benchmark results using full checks on all system calls. 35
4.2 Micro benchmark results using full checks on all system calls 36
4.3 Buffer size benchmark. Micro benchmark of the `read` system call. 38

List of Tables

4.1 Micro benchmark results of synchronization overhead. 38
4.2 Overhead of benchmarks on i7 setup 39
4.3 **Side-channel prevention**. Confusion matrix describing whether we detect timing attacks before information leaves the system. 42

Introduction

Memory errors are one of the most widely-used attack vectors on today's systems. Despite numerous attempts to mitigate these errors by use of modern tools and languages, memory errors, such as buffer overflows, still remain in a lot of legacy and high-performance software. High-performance code usually relies on having less overhead by using low-level languages such as C. These programming languages still rely on hazardous programming practices, such as manual memory management, that make it easy for developers to accidentally introduce memory errors. In [1] Van der Veen & al. give a broader overview of memory errors.

Recent efforts have put focus on mitigating such exploits by introducing additional security features, such as stack canaries[2], taint analysis [3], bounds checking [4, 5], or other run-time security features which (depending on the safety guarantees) may be quite costly performance-wise. Despite these efforts, many programs are still inherently unsafe. Thus, the focus has shifted to preventing the exploitation of such memory errors. Techniques such as Address-Space Layout Randomization (ASLR), as discussed by Schacham & al. [6], make it a lot harder to exploit memory errors by randomizing the address space. Nevertheless, techniques such as ASLR are easily circumvented by leaked pointers. Leaked pointers are one of the most commonly exploited attack vectors. Numerous efforts have been made to eliminate leaked pointers. However as can be seen in the work by Hund & al. [7], different side-channels can be used to leak information, making it possible to defeat ASLR.

Even if the application in question is safe, and proven to not leak any pointers, the Operating System is still a possible attack vector. The Linux Kernel, for example, consists of more than 15 million lines of code. On average there are about 6-16 bugs in every 1000 lines of code [8, 9], making the possible attack surface enormous. Even without explicitly being an exploitable bug, some errors can be used to leak information of the system state, which can be used to weaken defense mechanisms in a later stage of an exploit.

One active research area has been to utilize *multi-variant execution* (MVX) to detect errors in software. This technique runs different variants of semantically similar applications in parallel, synchronizing their states at certain intervals. By verifying that their state is still consistent, we can detect abnormal deviations from the normal control-flow. This technique was developed to detect possible bugs in mission-critical systems. Besides detecting errors in programs during testing, MVX has been applied in a more targeted-sense to detect exploits at run-time. In 1988 Joseph and Avizienis [10] first mentioned the idea of applying MVX to detecting exploits. By synchronizing programs at certain intervals, we can check that their execution path is still consistent. If an inconsistency is detected in a replica, we know that it has diverged from the expected execution path, and thus, has been exploited.

The challenge in MVX has been performance. Synchronizing the replicas on every instruction is infeasible, and not always necessary. Thus, most implementations only synchronize on external calls (i.e., system calls). Despite only synchronizing on these calls, a lot of information has to be shared when checking that the replicas are consistent. Another challenge has been to keep the replicas from *diverging* due to external I/O. By sharing *one single point of entry* (hereafter called a *barrier*) for data — such as keyboard I/O and network I/O — entering the system, we can ensure that the replicas do not diverge accidentally. The information going through this barrier is managed by a *monitor*, whose task it is to detect deviations and to keep the replicas from diverging. In [11] Koning utilizes hardware virtualization techniques to implement MVX in an efficient manner. One downside of such an approach is that these techniques are usually only found on newer, high-end CPUs. Also, they *limit the scope to code running with lower privilege levels.*

In this thesis, we present a kernel-level MVX implementation, *kMVX*, based on a modified multi-kernel fork of Linux (Popcorn Linux) [12]. By implementing a kernel-level Multi-Variant Execution Engine (MVEE), we can, in contrast to existing systems, also *detect* and *prevent* kernel-level exploits.

This thesis brings four major new contributions to the area of systems security:

- A design for running multiple, semantically equivalent, variants of a Operating System Kernel on a single system.
- Several *variant generation strategies* that effectively prevent kernel-level exploits, with a particular focus on preventing information disclosure.
- kMVX, a prototype implementation of kernel-level multi-variant execution environment with full Linux compatibility.
- A comprehensive detection mechanism for *side-channel-based* timing attacks, with no *false-positives*.

Chapter 1

Background

1.1 Kernel vulnerabilities

Due to the nature of large code-bases, such as the Linux kernel [13], it is inevitable that vulnerable code slips into the system. Even more so, it is quite common. In 2016, there were 96 registered CVE (Common Vulnerabilities and Exposures) in the Linux kernel alone [14]. These vulnerabilities range from *code execution-* to *denial-of-service-* and *information disclosure* vulnerabilities.

1.1.1 Types of exploits

In 2011 Chen & al. identified four major groups of vulnerabilities [15]:

1. *Memory corruption*
2. *Policy violation*
3. *Denial of service*
4. *Information disclosure*

In the past, several defenses against these types of vulnerabilities have been presented [1]. In recent years traditional *remote code injection exploits*, making use of buffer-overflows, have become obsolete due to the introduction of data *execution prevention* (DEP) techniques. Features, such as Intel NX [16], enforce a policy of making writable data non-executable. On the other hand, attacks such as *return-to-libc* and other *Return-Oriented-Programming* attacks overcome such techniques by executing existing memory-mapped code which necessarily has to be executable [17].

A common defense against ROP-exploits is *Address-space layout randomization* (ASLR). ASLR mitigates many of these attacks by randomizing the layout of a program in memory. For example, if an attacker does not know where executable code resides in memory, he or she cannot (easily) redirect the code-flow to a known location, making ROP-based attacks harder to execute successfully. Similarly, Kernel Address-space layout randomization (KASLR) has been applied to prevent similar exploitation techniques against the Operating System (OS) Kernel. There, however, exist several known weaknesses in previously used ASLR implementations [6]. Fine-grained ASLR makes circumventing these defenses harder [18, 19], since the code is split up into smaller blocks and shuffled around. Due to the prevalence of the `x86_64` architecture, the available virtual address space has become much larger, allowing for more bits of entropy when randomizing the address space. Fine-grained ASLR has also been applied to Operating System Kernels [20].

However, if the **attacker manages to leak a pointer**, ASLR defenses may be circumvented [21], allowing for existing attacks — that are normally mitigated by ASLR — to still be exploited, despite ASLR being enabled. In [22] the authors present an attack that makes it possible, due to *memory disclosure vulnerabilities*, to execute a *Return-Oriented-Programming* attack on an ASLR-enabled target. As we can see, the source of many modern *exploits*, both in user-space and kernel-space, is *information leakage*.

1.1.2 Information leakage

A popular class of vulnerabilities commonly found in the kernel are *information leaks*. Vulnerabilities in this category give away some confidential information that can enable an attacker to exploit the system.

Information disclosure vulnerabilities can go unnoticed for a long time and may be very simple to exploit once found, but very hard to detect. For example the vulnerabilities *CVE-2016-4569* (see Listing 1) and *CVE-2014-1444* can be used to leak contents from the kernel stack with the `ioctl` system call due to uninitialized data [23, 24]. By leaking kernel pointers with vulnerabilities like these, an attacker could possibly circumvent the — now commonly deployed — Kernel Address-space Layout Randomization (KASLR), allowing for more serious exploitation. Another possible way to exploit these vulnerabilities is to leak *confidential data*, such as cryptographic keys. As these kinds of vulnerabilities are often easy to introduce unintendedly, they are often encountered in large software project such as the Linux Kernel.

```c
// from sound/core/timer.c

struct snd_timer_tread {
         int event; // 4 byte padding
        struct timespec tstamp;
         unsigned int val; // 4 byte padding
};


int snd_timer_user_params(...)
{
//...
struct snd_timer_tread tread;
tread.event = SNDRV_TIMER_EVENT_EARLY;
tread.tstamp.tv_sec = 0;
tread.tstamp.tv_nsec = 0;
tread.val = 0;
//NOTE: padding is uninitialised, thus may contain old stack contents

//... eventually causes the call
copy_to_user(usr_buffer, &tread, sizeof(struct snd_timer_tread));
// which leaks uninitialized data to userspace
}
```

Listing 1: **CVE-2016-4569**. Example of information leakage vulnerability. These kinds of bugs are very common.

1.2 Multi-Variant Execution

Traditionally, *Multi-Variant Execution* (MVX) has been a development tool for finding bugs in mission-critical software. By running two semantically equivalent programs with small, non-observable to the outside, differences in them, one hopes to find bugs due to the divergence caused by the differences. Projects with large budgets can have two teams developing two versions of the same piece of software, allowing for MVX testing on these two variants. Recently MVX has been applied to detecting and preventing exploits [25–28].

Modern multi-variant execution engines (MVEE) consist of two parts: a *monitor*, the component running the different variants and comparing their output; and a *variant*

generation strategy. The monitor component affects the performance the most — since it has to compare the output — while the variant generation strategy determines whether a divergence can be found at all. A common strategy used by a majority of monitors is to interpose on *system calls*, since this is the primary way for a process to communicate with the outside world.

To prevent accidental divergence, we must ensure that the environment of the variants is consistent. It is, thus, the monitor's responsibility to prevent any divergence in the variants due to influence from the outside world. For example, `gettimeofday()` may be called at slightly different times in the variants (due to the variants having different number of instructions, or simply due to differences in scheduling). The monitor must ensure that all variants get the same return value for that particular call. Often a *master* is selected which actually performs the system call, supplying the return value to the replicas. The system call is then *replayed* on the *replicas*.

Variant generation

Historically, variants were considered different programs that perform the same task (for example GNU vs. BSD version of UNIX tools). Since writing two versions of each program or library is infeasible for real-world projects, the common approach is to introduce smaller changes between the variants. In the context of detecting exploits, one popular approach, as proposed by Cox & al. in [29], is to simply vary the memory layouts by using different memory allocators in the replicas.

Here we list a number of different variation techniques, as discussed in [30]:

- **Address-space layout variation**. In this category many variations can be introduced that alter how the program accesses memory. For example, a simple variation is to simply use different memory allocators in the variants or to enable Address Space Layout Randomization (ASLR). Other examples in this category include reversing the stack (i.e., let it grow up instead of down) [31], or changing the stack base. One recently popular approach has been to use disjoint address-spaces in the variants [29]. By utilizing disjoint (virtual) address-spaces, a pointer from one variant becomes invalid in all other variants, causing a segmentation fault if a pointer is dereferenced, enabling the monitor to detect a divergence.
- **Instruction set randomization**. By Randomizing the instruction set, for example, by `XOR`-ing all instructions with a known value at run-time, existing ROP attacks may be mitigated. In [32] such a variation technique is discussed.

- **Memory safety feature variation**. Different mitigation techniques can be enabled on the variants. For example, adding stack canary values [33] to prevent buffer overruns is a popular mitigation technique.
- **System call number randomization**. Usually the system call numbers are constant. However, much like the instruction set, these can also be randomized to prevent exploits that call certain commonly-exploited system calls [34].
- **Register randomization**. Some registers have a fixed meaning (e.g., the stack pointer in `ESP`). Another register can, for example, fill the functionality of the stack pointer register.

These different variation techniques can be used to create a number of different variants of the same program, without affecting the semantic of the program. In 2006 Cox & Al. [29] introduced the notion of running two semantically equivalent programs with different memory characteristics in parallel. By observing divergent behavior, it is possible to detect if a memory-layout-dependant vulnerability has been exploited. This breaktrough insight makes it easy to automatically generate different variants — just use different memory allocators for the variants. Automatically generating variants greatly reduces development time, while also allowing for a large number of variants (possibly covering a larger attack surface).

Monitor

Multi-variant execution has recently become a popular defense mechanism against a multitude of attacks [11, 35, 36]. These systems monitor user-space applications, synchronizing the variants on system calls by modifying the underlying operating system. Other approaches include patching the executable [37], so that a synchronization is performed at system calls (which may not be safe as the attacker could possibly tamper with the monitor); or using hardware-level process virtualization [11].

On of the biggest challenges in Multi-Variant Execution Environments (MVEE) is non-deterministic behaviour. In practice all non-deterministic behavior (save concurrency/ multi-threading) can be classified under I/O. For example user input from the keyboard, network packets, and hardware time fall under this category. To keep the replicas *consistent*, these kinds of I/O actions have to be synchronized across replicas. In Chapter 3 we will discuss in detail how we implement this kind of synchronization for multi-variant kernel execution.

The common strategy is to let a *master* replica handle all I/O, propagating the results to the *replicas*.

All these discussed MVEEs focus on detecting divergence in user-space applications. However, as we noted in Section 1.1, the kernel of the underlying operating system is as big an attack vector as the application running on top of it. Even more so, the attacks against the kernel may compromise the whole system, not just one application. In this paper we introduce a *kMVX*: a defense mechanism against kernel-level exploits.

1.3 Side-channel attacks

As defense mechanisms, such as Kernel Address Space Layout Randomization (KASLR) [20], have become a de-facto standard in modern operating systems, most current-day exploits have to take this into account. To be able to exploit a kernel vulnerability, an attacker usually needs to know the address layout (i.e., location in memory) of the kernel. Modern hardware has made it harder and harder to break *ASLR*. For example, current implementations of the `x86_64` architecture provide the user with a 48-bit address-space, giving ASLR a huge entropy to work with. The Linux kernel currently uses only 9 bits of entropy by default on `x86_64` [1].

Given the assumptions of enough entropy and no information disclosure vulnerabilities, KASLR can statistically guarantee a system to be safe from kernel-level attacks relying on absolute addresses. However, as noted by Hund & al. in [7], modern multi-user operating systems rely on shared hardware resources. By exploiting the fact that these resources are shared, an attacker may be able to leak information about the system, making KASLR easier to break. One hardware feature that is shared between all users of a system are the different memory caches. By measuring the access time to a cache, an attacker can detect whether there was a cache *hit* or *miss*, thus allowing the attacker to determine what memory has been accessed by other users of the system. By exploiting hardware features, such as the *Translation Lookaside Buffer* (TLB), an attacker can deduce what areas of memory have been mapped, thus circumventing KASLR without actually breaking the assumptions of KASLR.

Recently many shared hardware features have been used to leak kernel base pointers, thus breaking KASLR. In [38], the authors exploit a surprising behaviour of the Intel Transactional Synchronization Extension (*TSX*) that by-passes the kernel when a transaction fails due to a page fault. In [39], Gruss & al. exploit the *prefetch* instructions, allowing them to leak timing-info on arbitrary memory locations. In 2016 Schwarz & al. demonstrated an attack that does not rely on CPU features, but rather on intrinsic features of DRAM [40]. By reverse engineering the mapping of *physical addresses* to *virtual addresses* by means of profiling accesses to consecutive pages, the authors can

[1] `http://cateee.net/lkddb/web-lkddb/RANDOMIZE_BASE.html`

spy on memory access behavior of other users of the system (for examples the authors demonstrate how to spy on key-presses in the Firefox web-browser of a target user).

Due to the numerous ways of exploiting shared hardware features, leaking kernel information through these side-channel attacks has become a very urgent threat. Even more so, modern web-browser allow attackers to use high-precision timers to exploit these vulnerabilities [41], making these attacks a serious attack vector against every-day users.

Recent work has focused on mitigating these side channel attacks [42–44]. By using performance counters [42] found on modern CPU's, abnormal memory access behavior can be detected based on behavior models. By using statically trained models [42], or more dynamic machine-learning models [43], these side-channel attacks can be detected at an early stage. Usually these defense mechanisms slow down a process, or kill it when they detect abnormal behavior. A problem with these approaches is that there may be *false-positives*, thus either slowing down or in the worst case killing a totally benign process.

In this paper, we will utilize existing side-channel detection mechanisms [44], to detect possible side-channel attacks on the system. In contrast to existing systems, we will not stop the possible exploit immediately, but rather start profiling the possible malicious accesses by means of *multi-variant execution*. This approach removes the possibility of *false-positives* for the side-channel detection mechanisms, while providing the same safety guarantees.

1.4 Multikernels

Multikernels has in the past been a very active research area. The goal of a multikernel Operating System is to give the user the illusion of using one system, while the operating systems runs on a distributed system (either on multiple cores on the same node, or even on multiple networked nodes). Perhaps the most famous multikernel Operating System developed is Amoeba [45]. Multikernel Operating Systems never managed to get popular mainstream traction, due to slow communication between nodes. Due to the increased popularity of many-core hardware in commodity systems, a new interest in multikernel Operating Systems research has emerged. The Barrelfish Operating System [46], is a recent effort by MIT and Microsoft Research in renewing operating systems research for multi- and many-core systems. One of the goals of Barrelfish is to make better use of parallel hardware, eliminating blocking locks in the kernel found on unikernel systems. The downside of Barrelfish is that it is primarily a research-oriented Operating System

barely used in production. In 2013 researchers at *Virginia Tech* developed a multikernel port of the Linux kernel [12], called Popcorn Linux, giving the user benefits of better parallelization, while also not breaking user-space for existing applications. In principle, all normal Linux binaries can be run on Popcorn Linux. Popcorn Linux has also been applied for improved fault-tolerance in systems. FT-Linux [47], a port of Popcorn Linux, add state machine replication to the kernel in Popcorn Linux, allowing a replica to take over if the kernel if the master fails. In this paper we will utilize FT-Linux to implement kernel-level multi-variant execution.

Chapter 2

Design

In this chapter we present **kMVX**, a *multikernel-based multi-variant execution environment*, with the goal of preventing kernel-level exploits. We will give an overview of the design of *kMVX* and the rationale behind these design decisions. By utilizing a multikernel design, we can run multiple variants of a kernel in parallel on the same hardware. By implementing *state machine replication* on top of this multikernel Operating System, we can force the replica's to execute the same programs in parallel. By introducing small *variations* in the different kernels, we can detect malicious divergence paths caused by kernel-level exploits. In kMVX, the different *kernels* will be considered the *variants*, while a distributed *monitor* takes care of synchronizing the *variants*.

2.1 Threat model

As mentioned in [15] there are four major groups of kernel vulnerabilities:

1. *Memory corruption*
2. *Policy violation*
3. *Denial of service*
4. *Information disclosure*

Previously, MVX has been applied successfully to detect exploitation of user-space applications [11, 35, 36]. In this paper, we primarily focus on kernel-level exploitation. As argued in Sec. 1.1.2, most exploitation of the kernel relies on *information disclosure* as a first step to defeat existing mitigation techniques. Information disclosure in the kernel can also be used to leak *confidential information* like cryptographic keys. We,

thus, claim that *information disclosure vulnerabilities* are the most urgent category of vulnerabilities currently available. Hence, we will focus primarily on *information leak* prevention. Note that as a side-effect of our design, we can also prevent other types of exploits, like memory corruption attacks (this will be explained in Sec. 2.6).

We will assume that the attacker has *local access* to the machine, and thus has full control over a user-space application. We assume that the attacker wants to leak sensitive information from the kernel by interacting with the Operating System in a malicious manner, exploiting an existing vulnerability in the kernel. For example, one common scenario is that an attacker tries to defeat *KASLR* by leaking a pointer from the kernel (e.g., using vulnerabilities such as *CVE-2016-4569*). We want to prevent sensitive information leaving the system (e.g., via the network). As the only way for a process to communicate with the outside world (ignoring covert communication side-channels [1]) is the Operating System, we have full control over the information leaving the system.

2.2 Kernel vs. virtualization vs. hardware MVX

Multi-variant execution engines are usually implemented using one of the following three ways:

1. **Operating System level**. The operating system (in particular the way system calls are handled) is modified to enforce state replication monitoring between processes. The monitor is implemented in the kernel, allowing access to all processes.
2. **Virtualization level**. Hardware-based virtualization features are used to run variants. When system calls are performed, a subsystem performs the monitoring. The benefit is that the monitor does not need to run in kernel mode, reducing the attack surface.
3. **Hardware based MVX**. In mission critical systems, custom hardware (e.g., FPGA's) can be designed to perform MVX. For most use-cases this is too expensive and cumbersome.

Since hardware MVX is out of the question, we are left with virtualization and kernel-level implementations. Since we want to run multiple kernels in parallel rather than normal applications, there are some problems with applying existing virtualization techniques for kMVX. First of all, we need to compare our synchronization points. In a

[1] One such ingenious side-channel is load-based covert process communication in a virtualized environment [48]. Here Okamura & al. can succesfully perform inter-process communication without using the operating system.

virtualized environment this would be the *Virtual Machine Monitor* (VMM). Implementing this would be possible. Modern VMMs have a minimal overhead when running multiple virtual machines. The problem lies in that for kMVX, the variants run almost synchronously, performing the same tasks, and **requesting the same resources** from the VMM. This would introduce a bottleneck where the variants almost always will have to wait for the VMM to get the resource. If the VMM ensures a *fair-share policy* for the resources, the variants will most certainly trigger a race for the resources, context-switching the resources a lot between the variants. Also, the problem of finding common points (hereafter called system *barriers*) where information enters the system (*sources*) and leaves the system (*sinks*) is hard, as there are many resources that VMMs manage.

Because of these obstacles with virtualization, for kMVX we opt for a kernel-based implementation of kernel multi-variant execution. By utilizing multi-core hardware by using a multikernel design, we can run multiple variants in parallel without the problems encountered with virtualization. Also, the kernels have very clear *sources* and *sinks*, as will be discussed in the next section.

2.3 Kernel-level MVX

Multi-variant Execution Systems (MVEEs) consist of two parts: a *monitor* and a *variant generation strategy*. The monitor continuously checks the variants, detecting any divergent behavior. Thus far, MVEEs have been used to detect possible exploitation attempts in *user-space* applications, running on a single operating system. We propose applying this very same technique to *kernel-level* exploits. Applying this to kernels efficiently is a hard task, as the total number of possible *sources* and *sinks* is much larger than for user-space applications. There are many *sources* in the kernel. Everything from input originating from user-space communicated via system-calls, to inputs from hardware can be considered a *source*. Similarly, at these same points information can leave the kernel. Any interaction with user-space or with hardware (such as network ports) where the kernel writes information, is considered a *sink*. To prevent information leaks, it is important that our design incorporates hard synchronization points at *sinks*.

By using a multikernel design we can run multiple variants of a kernel on the same hardware. This is, however, not without its problems. Since we run on the same machine, the different kernels cannot access the hardware at the same time. Also, we need to *monitor* all hardware I/O. For sake of simplicity and consistency, one kernel will be elected *leader*. The leader acts as *monitor*, communicating with the outside world. All other kernels, or *replicas*, wait on the master for data from sources when necessary. Note that the replica kernels run asynchronously in parallel, thus they do not have to

wait for other replicas besides the master; they can continue executing at full speed as long as the master has executed all the I/O calls so far. The only exception when a barrier is enforced, is when a *sink* is reached. Since we stated in Sec. 2.1 that we want to prevent information disclosure, all variants have to reach the *sink barrier* before the master continues executing. In Fig. 2.1, a high-level overview of what the multikernel design looks like is shown.

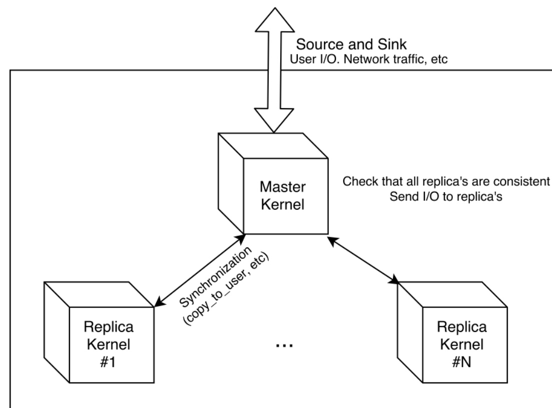

FIGURE 2.1: kMVX design overview. The master kernel communicates with the outside world, while the replica's verify that their state is the same as the master's.

2.4 Monitor

The monitor in kMVX is a distributed system, consisting of the monitor at the master's and the monitors at the replicas' ends. The monitor at the master takes care of sending all the results of hardware interaction to the replicas. To parallelize as much as possible of the actual monitoring workload, the monitor at the replica's end checks its monitor parameters are consistent with the master's. Another alternative would be to have the master do all checks which arguably would be more secure due to the fact that a compromised replica could fake its answer (see Section 5.3 for a discussion on how to improve the security). However, doing all checks on the master would introduce *load imbalance* proportional to the number of replicas that the master has to check.

For example, when a network packet is received, the master takes care of communication with the *Network Interface Card* (NIC), sending the contents of the packet to all the replicas. The master is free to continue executing, communicating with hardware freely; it does not have to wait for replicas at *sources*. However, when the goal is to prevent information leaks, the master has to enforce a hard barrier at *sinks*. The master waits until it has received confirmation that there is no divergence from all replicas before continuing its normal execution.

As the replicas are not allowed to communicate with *sources* directly, they have to wait on the master to reach the *source barrier*, thereby obtaining the source data from the master. Meanwhile, at *sink barriers*, the replicas do not have to wait; they do, however, still have to wait for the master to finish that particular sink barrier before the next *source barrier* can be completed.

The question now becomes, how do we cover all *sources* and *sinks*? In our prototype implementation (described in Chapter 3) we have opted for a simplistic implementation that only considers local attacks (we only monitor information exchange between kernel- and user-space). But for completeness, some ideas on sources and sinks are discussed in the following sections.

2.4.1 Sources and sinks

In Operating System Kernels, many points can be considered *sources*. All points where information can enter the kernel are considered a source. We have identified two categories of kernel entry points: 1) communication with user-space by means of *system calls* and 2) *hardware I/O*. Hardware I/O may for example be user input from the keyboard, or reading the time from a Real Time Clock, or even incoming network packets.

The first category, system calls, is the only way for a user-space to communicate with the kernel. In other MVEEs, systems calls are also the primary events being monitored. Usually MVEEs have to be aware of the semantics of the system calls (e.g., what type does it return? does it write to a buffer?), so that the return values can be compared. In our design we could also approach monitoring of system calls in the same manner. However, since we care about the kernel, rather than the user-space application, we do not have to care about the semantics of the system call. We are simply interested in the information entering the kernel. In most operating systems, all copying of user-space data is done through one single entry-point: `copy_from_user`. By intercepting all calls to `copy_to_user`, we can effectively monitor all information entering the kernel from user-space.

The second category, hardware I/O, is a bit harder. Since the actual information entering the system depends on the hardware and driver, there are many different ways information can enter the kernel. Luckily, Operating Systems like Linux usually provide an abstraction. For example, network communication is all done using `sk_buffers`, which makes it possible to find functions covering a large area of entry points. Other hardware I/O sources may be the file-system, USB-devices, etc. These devices all have some kind of abstraction in the kernel, making it relatively easy to find appropriate sources.

As with sources, *sinks* can be put into the same two categories. The sink to user space is the entry-point `copy_to_user`, which is the opposite operation of `copy_from_user`. Likewise, finding appropriate hardware I/O barriers for sinks is harder than finding a good barrier for user-space communication.

Another point to note is that **whether something is considered a *source* or *sink* depends on the point-of-view** of the MVEE. If our goal is to utilize kMVX on, for example, only a subsystem of the kernel (say a kernel module), then all information leaving that subsystem leaves through a *sink*. In the next chapter we will apply kMVX to detecting information leaks through means of **local attacks**. We want to prevent user-space from leaking confidential information obtained from the kernel. As such, `copy_from_user` will be considered a *sink* rather than a *source*, since information **leaving** user-space is considered leaked.

2.5 Severity level manager

As the kernel contains many barriers, doing a hard-synchronization at all barriers would be infeasible from a performance perspective. Our design includes a *severity level manager*, allowing the system administrator to determine to give up some security for better performance.

For example, we may only wish to track a subset of all user-space communication or a subset of system calls. We propose a fine-grained mechanism, a *severity level manager*, that keeps track of which sources and sinks should actually be monitored depending on the threat model. For example, if local access is not part of our threat model, we may not care about user-space synchronization at all, since user-space may be considered a closed system that always has to talk with the kernel if it wishes to send information out of the system. Also, some frequently used system calls (like `getpid()`) may be considered relatively harmless, thus to gain performance, it might be of interest to disable monitoring of such system calls. Furthermore, some local processes in the system may be totally trusted, thus it would make little sense to monitor the communication between the kernel and these processes. It is desirable to affect the performance of the system as little as possible. It makes sense to only monitor barrier to the kernel that may be of interest. Using a severity level manager, it is possible to only monitor barriers of selected processes (e.g., untrusted user processes). To reduce the number of monitored barriers even further, we may only monitor processes that we consider exhibiting *abnormal behavior*.

2.5.1 Abnormal behavior detection

Our severity level manager contains a sub-system, called the *abnormal behavior detector*. This sub-system is designed in a extendable manner, allowing *detectors* to be added easily. Abnormal behaviour may be defined as everything from suspicious access to memory (e.g., many page-faults), to spurious use of system calls (e.g., too many similar system calls). Since our threat model states that we focus on information leaks, we want to include possible attacks using hardware-based side-channel attacks.

2.6 Variant Generation

The second major component of a MVEE is the variant generation technique. Here, we propose a few different variation techniques that can be applied to generating kernel variants. The following techniques can all be used in combination:

- **KASLR**. Since most modern operating systems include kernel-level address-space layout randomization, we will keep this enabled for our multikernel design. On start-up the different kernels will randomize their base-address within their designated memory area.
- **Memory allocator variation**. We propose running modified versions of the kernel's memory allocator. For example, in Linux, the SLAB allocator can be modified to exhibit different behavior, resulting in different address-space layouts. In Chapter 3 we will discuss a modified type-safe variant of the Linux SLAB allocator.
- **Stack layout variation**. The layout of the kernel stack can be modified. On compile-time the behavior and layout of the stack can be modified. For example, [31] introduces a modification to GCC, to make the stack grow in the reverse direction, preventing an attacker from overwriting return-addresses in all variants.
- **Disjoint virtual address-space**. This technique ensures that the kernels in the different variants have totally disjoint virtual address spaces. The technique is similar to the one put forth by Cox & al. in [29], with the exception that we are talking about kernel-space instead of user-space. By ensuring that the (virtual) address-space of the variants is disjoint, a valid pointer will deterministically be invalid in other variants. If an attacker somehow tries to de-reference a pointer, this will cause a page-fault in the other variants, thus we are able to detect the malicious behavior. See Figure 3.4 for an overview of how the virtual address-space of the kernel can be partitioned such that it is disjoint.

Of the above mentioned the three first techniques are *probabilistic* defenses — we hope that the difference in memory-layout causes a divergence that is later detected by our monitor. For example, varying the stack layout *may* cause divergence in the data leaked when exploiting a uninitialized data vulnerability on the stack. Nevertheless, there are no strong guarantees that the divergence of the stack layout actually causes different data to be leaked. Likewise, KASLR opportunistically prevents de-referencing of pointers in the variants — it may happen that the pointer is valid in all variants and de-referencing it has no visible effect outside the barriers, but might have internal side-effects. The last technique *deterministically* prevents the attacker de-referencing pointers. In contrast to KASLR, this technique ensures that if an attacker de-references a pointer, it will be considered invalid in all other replicas. This gives us a very strong defense against memory disclosure vulnerabilities. In the next chapter we will discuss how we apply these variation techniques to our prototype Linux-based implementation of kMVX.

Chapter 3

Implementation

In the previous chapter, we discussed the design of our kernel-level multi-variant execution engine. In this chapter we will discuss how we apply this design to a modified multikernel port of the Linux kernel. By basing our implementation on the Linux kernel, we not only demonstrate that our design works in theory, but also show that it can applied to existing real-world systems. By preserving compatibility with Linux user-space, all existing software can still run on our system even without the need of re-building the software. This is of particular interest for users that run legacy or proprietary software where the source code is not available. In Chapter 4 we will evaluate our implementation with a number of benchmarks and security exploitation scenarios.

As stated in Section 2.1 we primarily focus on *local attacks*. Therefore, we limit the scope of our prototype implementation to the *userspace-kernel barrier* (i.e., we only monitor data-transfer to- and from user-space). With some work the prototype can be extended to include hardware I/O barriers. As we do not monitor hardware I/O we consider information **leaving user-space to be leaked**. From the point of our defense, we consider `copy_from_user` a *sink*, while `copy_to_user` is a *source*. Note that, while this approach may look a lot like a user-space MVEE, we actually **detect divergence** in information entering user-space through means of system calls, allowing us to detect **kernel-level exploits**. Previous MVEEs execute all system calls **in the same kernel**. Du to there being no divergence between the kernel handling the system calls, information leaks, such as leaked pointers, may go unnoticed.

We base our implementation on *Popcorn Linux*[12], a multikernel port of the Linux Kernel developed at Virginia Tech [1]. The current version is forked from Linux version 3.2.14. By utilizing this existing multikernel port, we can with relative ease implement our kMVX system on top of it. We demonstrate a prototype implementation for the

[1] `http://vt.edu`

x86-64 architecture. For the case of simplicity, we only use one master and one replica. However, our implementation can be altered relatively easily to run more than two variants.

3.1 Utilizing Popcorn Linux for MVX

3.1.1 Multi-kernel design

Popcorn Linux introduces a multi-kernel design based on the Linux kernel. The benefit of this approach compared to other multi-kernels, such as Barrelfish OS [49] or Amoeba [45], is that Linux user-space is preserved; all applications that run on Linux still run on Popcorn Linux. Each kernel has its own disjoint address space and its own sets of cores to work with. However, the rest of the hardware is shared. As we want to ensure that only the master replica communicates with hardware (as discussed in Sec. 2.3), only the *master kernel* communicates with I/O channels directly.

On essential component that Popcorn Linux introduces, is an inter-kernel message passing layer, allowing the different kernels to communicate with each other. For this purpose, the message passing layer utilizes a shared in-memory *ringbuffer*.

A more comprehensive description of Popcorn Linux can be found in [12]. Popcorn Linux has been used for many research projects in the area of heterogeneous computing and fault-tolerance on many-core hardware [2].

3.1.2 FT-Linux

FT-Linux is a fault-tolerant operating system based on Popcorn Linux [12]. It implements Primary-Backup replication, wherein a replica kernel can take over the work of the master in case of hardware failure. The Primary-Backup replication is achieved through full software stack replication for all applications running in the *ft-linux* kernel namespace [47]. We implement kernel-level multi-variant execution by utilizing this existing full stack replication. The benefit of using FT-Linux is that it already includes replication of user-space applications and the network stack, as well as replaying certain system calls that may cause divergence in the replicas. Furthermore, it includes fail-over mechanisms allowing us to elect a new master, should the master crash.

Since FT-Linux is designed for replication it already makes sure that the state of the replica kernels is equivalent to that of the master kernel. Some system calls are inherently

[2] See `http://www.popcornlinux.org/index.php/publications` for a list of projects.

non-deterministic and need to be replayed to keep the replicas in sync. For example, the `gettimeofday` and `time` system calls are replayed, ensuring that the user-space applications in the replicas do not diverge.

FT-Linux also provides a replicated TCP stack, based on the design presented in [50]. Internally, the network stacks are state-replicated. Information, such as sequence numbers and connection state are shared between master and replica. Despite the network stack being replicated, the communication with the NIC is only performed on the master. To keep the state between the variant consistent, the network system calls — including `accept`, `accept4`, `rcv`, `rvc_from`, `send`, `send_to`, `epoll`, `epoll_wait`, and `poll` — are replayed on the replicas based on the return values on the master.

A drawback with having the replicas replaying all *sources* by obtaining the return the values from the master, is that this reduces the attack surface significantly. When replaying from the master the possible vulnerable code is never actually executed on the replicas, rather, the replica skips running the calls itself and copies the result from the master. Especially hardware drivers, whose code is only executed on the master, are a major attack surface [15]. By replaying from the master, a vulnerability found in the master can be exploited without detection by the MVEE. Furthermore, replaying *sources* is expensive, as communicating the result to the replicas incurs some overhead.

For the sake of improving coverage (as well as for performance reasons), we have opted to use different file-systems for the master and the replicas. The replicas use a *ramfs*-based file-system, so that they do not have to communicate with a hard-drive controller. By utilizing separate file-systems we eliminate the need to replay disk I/O calls.

One downside of this approach is that the state of the file-system has to be identical when starting the system. For our benchmarking purposes, we ensure that the file-systems are consistent on boot. In 5.3 we discuss a possible, more robust, solution that would not require replaying all filesystem operations from the master.

Another source of undesired divergence between the variants is caused by non-determinism due to multithreading. FT-Linux includes support for deterministic multithreading by adding two system calls (`__det_start` and `__det_end`). All calls to the *Pthreads* API between these two system calls will be serialized and replayed on the replicas.

3.2 Monitor

In FT-Linux the replicated system calls are replayed in the replicas. Since they are replayed, the replicas have to wait for the master to finish executing that particular

system call, before continuing their execution. This waiting, understandably, introduces some overhead. Note that only a small subset of all system calls are replayed. For example `gettimeofday`, which may return different values if the replicas are even slightly out of sync, is replayed from the master. Systems calls that do not need to be replayed are executed locally on all the replicas in an asynchronous parallel manner.

Besides the system call modifications introduced in FT-Linux we add some additional modifications to the implementation of a select group of system calls. Since FT-Linux focuses on fault-tolerance in servers, minor divergence is not a major concern. For example, `getpid` may cause some divergence if used in conditionals. We modify the `getpid` system call to return a namespace-unique identifier (this is already provided by FT-Linux) that is the same across replicated processes in the variants. Another source of divergence is the `rdtsc` instruction. This source of divergence can be disabled by setting the `TSD` flag in the `cr4` register of all replicated processes. In Section 5.3 we discuss how to enable `rdtsc` while keeping the variants from diverging.

Apart from replayed system calls, another synchronization point is the actual monitor *barrier*. As discussed in Section 2.4.1, the *barriers* between user- and kernel-space are very clear. The two main functions in the Linux kernel that allows communication between these two are `copy_to_user` and `copy_from_user`. As discussed, we limit the scope to local attacks. As such, we consider information that leaves user-space to be leaked. Thus, we will consider `copy_to_user` a *source*, and `copy_from_user` a *sink*.

3.2.1 Copy-to-user synchronization

In Linux, all copying of information from *kernel-space* to *user-space* goes through the `copy_to_user` function. At this synchronization point, the replicas wait for the master kernel to send its `copy_to_user` buffer value.

We insert our instrumentation by modifying the `copy_to_user` function (or rather the `__copy_to_user` function, since the `read`-system call uses this alias) defined in `arch/x86/include/asm/uaccess_64.h`. If the current process context is in the *ft-linux* namespace, we call our custom `mvx_copy_to_user` monitor-function. A simplified version of the monitor function can be found in Listing 2. This approach allows us to control which calls to *monitor* in a fine-grained manner. Inside the `mvx_copy_to_user` function, our *severity level manager* further decides what security measures to enforce. For example, if we suspect that an application tries to make use of hardware-based side-channels, we can enforce lockstep *copy-to-user* synchronization.

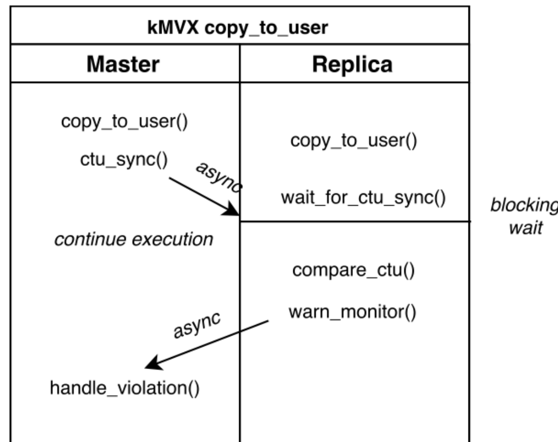

FIGURE 3.1: `copy_to_user` synchronization in kMVX.

In each kernel some additional information is kept so that we can identify the *barrier points*. The per-task information struct, `task_struct`, in `include/linux/sched.h` is augmented with a barrier counter. We use this barrier counter for identifying messages when communicating between the master and replica.

All messaging between the kernels is dispatched using the `pcn_kmsg` system in Popcorn Linux. The message between the kernel contains some necessary information, identifying the message, and a content buffer. The content buffer contains the data to be verified by the monitor (i.e., the value of the buffer provided to `copy_to_user`). Each time the process reaches a barrier that the severity level manager deems necessary to synchronize, the counter is incremented. Note that the value of the counters in the replicas can at a given moment vary greatly. However, when a hard-sync barrier is reached (as with `copy_from_user`), the counters will be equal in all replicas (unless, of course, a divergence has occurred)

When a message arrives at the destination kernel, the kernel executes a callback handler function (registered using Popcorn Linux's `pcn_kmsg_register_callback`) in an asynchronous manner. In this handler, we save the *barrier info message* in a hash table, where the key is a combination of the current process id, current system call, and barrier counter. This information can then easily be retrieved when necessary. Using a hash table allows us to do information lookup with an average $O(1)$ time complexity.

For a representation of how the copy-to-user monitoring works, see Figure 3.1. In this figure, the master sends its information to the replica, that in turn compares the values against its own return values. If the replica detects a divergence, it warns the master. The only blocking synchronization point is in the replica when waiting for the master's values.

```
// Global context info
// this->ctu_id;

void mvx_copy_to_user(void *dst, void *src, size_t len)
{
    if (node_is_master()) {
        struct mvx_mp_message *msg;
        msg = construct_mvx_message(/*context variables here*/, len);
        // Copy ctu content into buffer
        memcpy(msg->ctu_content, src, len);
        send_to_replicas(msg);
    } else {

        struct mvx_mp_message *msg;
        // Wait for primary
        wait_for_ctu_message(&msg);
        // Compare ctu
        compare_ctu(msg, len, src)
    }

    this->ctu_id++;
}

void compare_ctu()
{
    // 1. Compare syscall id
    // 2. Compare ctu length
    // 3. Compare ctu contents

    if (flagged) {
        // Handle kMVX policy
    }
}
```

LISTING 2: **Simplified copy-to-user synchronization code** — We detect exploit-attempts making use of vulnerability CVE-2016-4569, when comparing value of the buffers.

3.2.2 Copy-from-user synchronization

Analogous to `copy_to_user`, `copy_from_user` is the point of all data entry into the kernel from user-space. Since our goal, according to our threat model, is to **prevent information leaks**, the copy-from-user barrier is the most critical point of information leakage in the system. Since we do not track other sinks in the kernel, copy-from-user is our primary sink. If confidential information leaves this barrier, it can be considered leaked.

Since we do not enforce a hard-barrier at the copy-to-user barrier, a replica which is being exploited can already be compromised while still continuing its execution. To prevent information leaking from the system, we enforce a hard synchronization barrier at the `copy_from_user` barrier. By having the master kernel wait on an acknowledgement message (containing the result of the check performed on the replica) from each replica before sending out the information into the open world. By enforcing such a *hard barrier* we ensure that all *divergence checks* have been executed before information leaves the system.

In Figure 3.2 we show how the `copy_from_user` synchronization is executed. In the figure, the master sends its information to the replica. The replica, in turn, compares the values against its own return values. The master waits for a reply from the replica before it continues, preventing information leaks from leaving the user process.

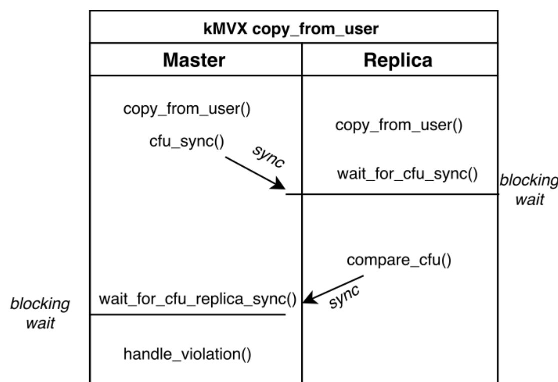

FIGURE 3.2: `copy_from_user` synchronization in kMVX.

3.2.3 System call modifications

Some system-calls on modern systems use the *virtual dynamic shared object* (vDSO) mechanism to execute some system calls without imposing a kernel context-switch overhead. Since we want to instrument all system-calls, and since we have to trap into the kernel anyhow when communicating between replicas (eliminating the performance benefits of vDSO), we opt to disable the vDSO completely using the `vdso=0` boot parameter. Disabling the vDSO will decrease the performance remarkably for a number of commonly used system calls. In Sec. 5.3 we discuss a possible solution, improving the performance.

3.3 Severity level manager

Our system includes a *severity level manager* which selects which system calls to monitor and how to synchronize them. The severity level manager allows us to trade in security for performance. We can disable hard synchronization on `copy_from_user` and only enable this when *suspicious activity* occurs. When a *detector* flags a process as suspicious, we increase the severity level of that process, thus enforcing stricter checks.

The severity level manager runs in kernel mode on all replicas. Management of policies is done on the master node, which in turn sends out changes in policies to the replicas using the message passing interface in Popcorn Linux. If a replica kernel thinks that a process should get a higher severity level (e.g.,, due to detected abnormal behavior), it messages the master. The master, in turn, decides whether the policy should be updated. If it deems it necessary to update the policy, it send out a message to all replicas, instructing them to change the policy as soon as possible. For now, to avoid complexity, we only change the policy when we reach a hard-barrier.

We currently have three default policies:

1. *Divergence logging.* In this policy, we do not enforce hard *barrier* synchronization. Due to this fact, information may leave the system before a divergence has been detected. This policy may be used for logging purposes in mission critical systems, to reconstruct how an exploit happened.
2. *Information leakage.* The goal of this system is to prevent confidential information leaking from the system. We enforce a hard synchronization on all *sink barriers*, while the *source barriers* can be run asynchronously.
3. *Lockstep.* We enforce a hard synchronization at all *barriers.* This is the strictest — and costliest — policy.

3.3.1 Performance vs. security goals

Using the severity level manager we can trade in security for performance. For example, some system calls, such as `read` and `write` copy a lot of data between the barriers. In some instances it may be interesting to not actually monitor the contents of the buffers, but only the length of the buffers, eliminating a lot of messaging overhead (larger messages may have to be split up into smaller messages internally in Popcorn Linux's message passing layer). Our severity level manager can set these kinds of policies on a per-task, per-system-call granularity.

3.3.2 Activity Detectors

kMVX includes a mechanism to add *activity detectors*, small subsystems that can trigger a higher severity level if it detects *unusual behavior*. Unusual behavior can be anything that deviates statistically from normal behavior. In this thesis, we will focus mainly on hardware-level side-channels, such as *cache timing attacks*.

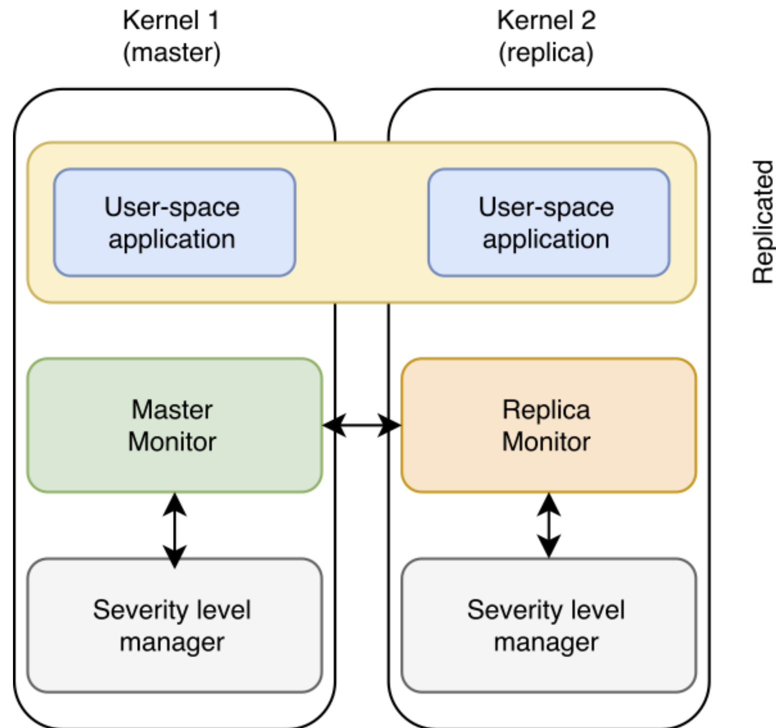

FIGURE 3.3: kMVX multikernel design. The kernels communicate through message passing.

3.3.3 Side-channel detection

We utilize previously published defense mechanisms to detect possible side-channel attacks. Our prototype classifies possible side-channel attacks by using *performance counters* to detect cache attacks. While this is only one of many side-channel detection measures, our system is designed to be expandable with many types detectors. Currently we monitor all processes in the replicated namespace. The side-channel detector is based on a side-channel detection framework presented in [44]. By reading performance counter values (such as the cache miss counter) of processes active in the system, we can classify possible side-channel attacks. More specifically, our current implementation only detects possible *cache-timing attacks*. We profile each process in the replicated namespace by continuously taking samples of the performance counters using the `perf` API in Linux. Using the 60 most recent samples we calculate our statistics. By looking

at processes with a *miss-to-hit* ratio of over 0.7, we can detect possible cache-attack attempts.

Suppose a cache-timing attack that reverse-engineers the mapped memory areas of the kernel (see [41]) were executed on all replicas. When detecting such activity, our policy manager enforces total lockstep-mode, allowing us to detect the slightest divergence. The benefit of this approach, compared to other side-channel prevention mechanisms, is that we will have no false positives. Suppose our side-channel detector classifies a benign process as a possible threat. We do not immediately kill the process or slow it down deliberately, but rather we start monitoring it. If the process behaves correctly (i.e., there is no divergence), we do not affect its life-cycle. As we will have no false positives, our side-channel detector is always *pessimistic*, assuming that a small deviation in behavior constitutes a possible attack. If we, however, detect a possible anomaly too late the attacker may have already leaked information before our protection mechanism kicks in. In Section 4.3 we will have a look at how efficiently our mechanism prevents leakage of information.

3.4 Variant generation

3.4.1 Non-overlapping virtual address space

As discussed in Section 2.6, we use partitioned virtual address spaces to make valid pointers from one variant invalid in all other variants. In Linux, the virtual address space of a process consists of user-space and kernel-space. The kernel is mapped in memory to prevent expensive cache flushes when trapping into kernel-level (i.e., by means of system calls).

The process' address space in Linux are split into different areas, all defined in the kernel. We have identified a number critical kernel memory areas that we definitely want to keep disjoint between variants:

1. **physmap** is large contiguous virtual mapped area that is mapped one-on-one with the physical available memory. Its primary use is to quickly allocate physically contiguous memory through *kmalloc*.

2. **vmalloc** space is used for obtaining virtually contiguous memory areas backed by fragmented areas of physical space.

3. **virtual memory map** consists of an array of page structs. This length of this array is equal to the number of physically available pages in the system.

4. **kernel text area** containing the executable code of the kernel.

5. **module area** the area where kernel modules are loaded.

We have modified the *size* and *locations* of these memory areas, as depicted in Figure 3.4. Most of these areas can be easily partitioned by modifying constants found in `arch/x86/include/asm/pgtable_64_types.h`. For our experimental setup, we modified these by hand — however, the process could be automated by a script. By modifying these values carefully for each replica, we ensure that each of these areas in every variant is disjoint. As the virtual address space of these areas are huge (in the Terabytes range), reducing the size is currently not an issue.

Making the kernel text area disjoint takes some more work, since these values are hard-coded in boot-time assembly files. To achieve disjoint kernel text areas, the boot code for the secondary kernel has to be modified. We modify the page table section of the secondary replica to zero out entries corresponding to the primary kernel, causing a page fault if the kernel text area of the primary replica is accessed. The modifications implemented here reduce the area for the uncompressed kernel text to 256MB, which should still be plenty to run any kernel version.

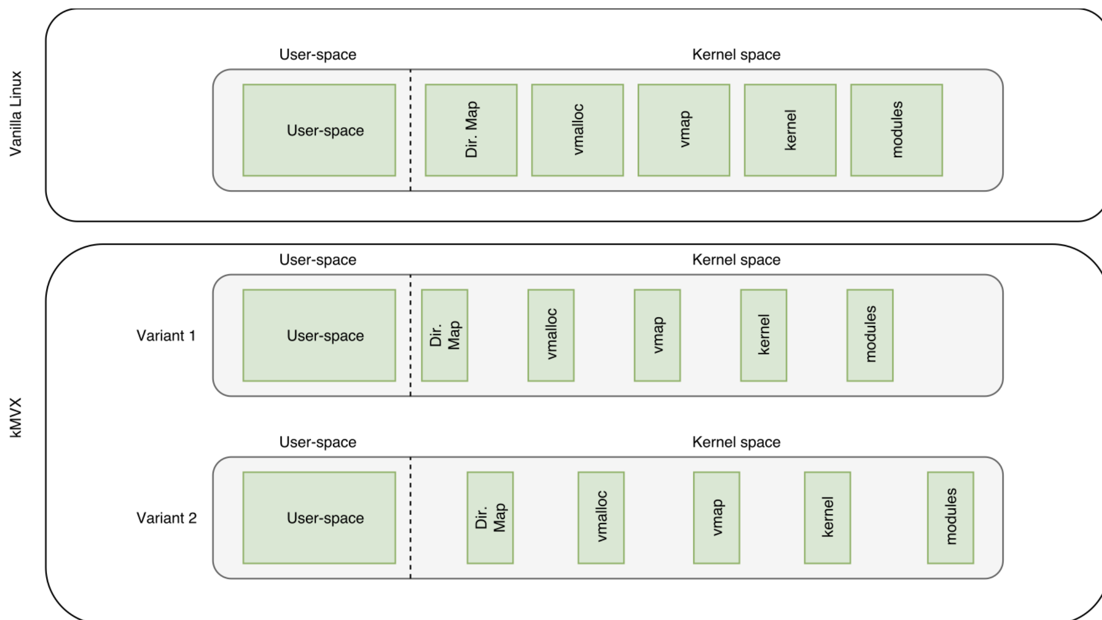

FIGURE 3.4: Partitioned virtual address space in kMVX.

3.4.2 Variation in memory allocators

Besides the *deterministic* variation gained by the partitioned address space, we also use some *probabilistic* variation techniques to modify the memory layout. By using a

different kernel memory allocator, a *type-safe SLAB allocator*, we increase the detection surface of exploits using *use-after-free vulnerabilities.* We replace the SLAB allocator in the replica kernel by an allocator that uses caches *according to type*, rather than the usual *per-size* caches. To create a per-type cache, we need to know some more information about the allocation (i.e., the type) than currently known by the allocator.

One option is to annotate the source code with type information. We have, however, opted for a simpler approach. By viewing the return address of the allocation (i.e., the instruction pointer of the `kmalloc` call-site) as the type, we can easily create a per-type cache. We map each call-site to a type, through a *hashing function*, dynamically creating a new cache on run-time when we discover a new type. One issue encountered is dynamic size allocation. If a call like `kmalloc(strlen(s))` is executed, the length of the allocation will differ between calls, and as such are not suitable for our type-based allocation pools. We modify calls like these to fall back to default size-based behavior.

While this approach does not always accurately map allocations of the same type to the same pool (several call-sites can allocate the same type, but with this approach they are considered different types), we argue that it will still create a vastly different heap memory layout than the default SLAB allocator. If desired, the *hashing function* — mapping types to pools — can be altered, allowing for easy, automatic variant generation of the memory allocator.

3.4.3 Stack format

By changing the format of the kernel stack in the variants, we can prevent some common stack attacks like buffer over-runs and on a best-effort basis try to detect uninitialized data reads. Randomizing the stack for a kernel variant can be achieved trivially at compile time by utilizing different compiler flags for the variants. Using `GCC` flags such as `-fdouble-stackvars`, `-freverse-stackvars`, `-frandom-stackpad` and `-fstack-protector-strong` we can introduce variation in the stack layout of the different kernels. Suppose, one variant has the `-freverse-stackvars` set. Chances are that if an attack manages to leak (a part of) a pointer in one variant due to uninitialized data (such as in *CVE-2016-4569*), the variant with the flag set will have different data on the stack, allowing us to detect a divergence when we perform our monitoring check in `copy_to_user`.

We have, so far, described the implementation of our prototype. We have discussed how we modify Popcorn Linux, and utilize its multikernel design to implement a simplistic kernel-level multi-variant execution engine. In the next chapter we will benchmark our implementations, and discuss how it can be used to prevent real-world exploits.

Finally, in Chapter 5, we will discuss some future improvements that can be made to our implementation to achieve better vulnerability detection and better performance.

Chapter 4

Results

4.1 Threat model

In our threat model we assume the attacker has access to a memory exploit, be it a buffer overrun, heap overrun, integer overflow, or anything in this category. We assume the attacker has local access to the system, i.e., the attacker has total control over user-space. We assume the system has modern safety features such as Kernel Address-space Layout Randomization (KASLR) enabled and all the safety mechanisms discussed in the previous chapter. Since we focus on preventing information leaks, we use the *information leakage* policy in our monitor for all our benchmarks.

Our goal is to prevent local exploits that make use of leaked information. Common attack vectors include leaking kernel pointers to defeat KASLR. We also consider, that an attacker may use hardware-based side channels to obtain this leaked information.

4.2 Performance evaluation

We benchmark our prototype implementation of kMVX using several micro- and macro-benchmarks. We argue that the selected benchmarks give an overall good image of the performance characteristics of kMVX, both in artificial worst-case scenarios, and in more realistic settings.

4.2.1 Experimental setup

We run all our benchmarks on a 2-core AMD Athlon X2 7750. We compare our implementation against FT-Linux and a vanilla (non-replicated) Linux 3.2.14 kernel. We

reproduce part of the setup used in [47], with the major difference that our system has fewer cores and less memory. We connect two machines with a 1GB/s local area network link. As can be seen in Figure 4.1, the benchmark over LAN (denoted *nginx(LAN)* does not saturate the system fully compared to running all network traffic over the *loopback interface* (denoted *nginx(LL)*). To be able to show results under a more utilized system, we include a benchmark where network communication is *link-local*. The downside of using a link-local approach is that it does not show real-world performance, as we eliminate the usual latency. We argue that the performance on a real-world network can only become better since the server is less saturated. In all benchmarks we run kMVX using one *master* kernel and one *replica* kernel.

We have also run the benchmarks on a 4-core i7 6700 with 16 GB of RAM with link-local network access, such as to saturate the system fully. However, due to unresolved hardware interrupt errors, we have not been able to run all benchmarks in this setup, thus these findings will not be reported here. We present a select number of benchmarks performed on this system. Note that FT-Linux is optimized for the setup described in [47], thus some hard-coded values may be sub-optimal for the i7 CPU.

We evaluate the performance of kMVX using the strictest policy preventing information leaks (i.e., we instrument all system calls and enforce a hard synchronization on *sinks*). We argue that this is the most restrictive policy (besides total lock-step execution), and thus shows our worst-case performance.

Micro-benchmarks

We benchmark our prototype implementation by running some micro-benchmarks on a selected set of system calls. We benchmark the performance of our micro-benchmarks using the Time Stamp Counter (`rdtsc`) to get an accurate high-resolution performance measure of the time for each system call.

- **Small buffer system-calls benchmark.** This benchmarks executes 1000000 consecutive calls to the `getpid` system call, calculating the average clock-cycle cost per system call.
- **Medium buffer system call**. This benchmark executes 1000000 consecutive calls to the *getcwd* system call, calculating the average clock-cycle cost per system call.
- **Replicated system call.** This benchmarks executes 1000000 consecutive calls to the `gettimeofday` system call, calculating the average clock-cycle cost per system call.

Besides these micro-benchmarks we also evaluate how comparing the contents of the data crossing the *barriers* affects our performance We benchmark the impact of the size of the buffer provided to the `read` system call. We benchmark how the size affects the total cost of the system call. For each size, *1, 2, 4, 8, 16, 32, 128, and 4096 bytes*, we execute 10000 consecutive reads and measure the difference in the Time Stamp Counter. This gives us a realistic view of how the performance is affected in real-world applications. To eliminate the very high overhead of disk reads, we utilize the `memfd_create` to create an anonymous memory-mapped file to read from.

Furthermore we evaluate how *hard synchronization points* affect the performance. We time what fraction of *instrumentation time* the master kernel has to wait for the replica at a `copy_from_user`-synchronization. Similarly, we benchmark what fraction of the time the replica spends waiting on the master for `copy_to_user` calls.

Macro-benchmarks

For our real-world macro benchmarks we use four benchmark suites with different characteristics, giving us a good overview of the performance in different settings.

- **Nginx with Apachebench**. We benchmark the Nginx webserver with 20000 request using 30 concurrent connections. The file requested is a 10kB static webpage. This setup is typical for a server. This test puts stress on the network system calls.
- **LMBench**. We benchmark system calls only using LMBench (i.e., LMBench's OS benchmark). This benchmark tests various system calls.
- **Memcached**, is a distributed general-purpose in-memory caching system, typically used as database caching layer. The operations in Memcached are memory intensive. We use the *memaslap*[51] benchmark suite to evaluate the performance of memcached. We run one single-threaded instance of Memcached with default settings.
- **TCPEcho**. A simple TCP server that echoes back the content sent to it. We send 10000 consecutive TCP Packets that are echoed back. This mainly stress-tests the network system calls.

4.2.2 Results

Macro benchmarks

Since our design is based FT-Linux, we primarily compare our performance to FT-Linux — as improving performance in FT-Linux would also improve the performance of kMVX.

As can be seen in Figure 4.1, not unexpectedly, we incur more overhead that FT-Linux. In the *nginx*-, *memcached*-, and *TCPEcho*-benchmarks our results are comparable to FT-Linux. In *lmbench* we get about twice the overhead of FT-Linux, which can be explained by the fact that we have to communicate between the kernels for all system calls, while FT-Linux only incurs communication overhead on a small subset of system calls. In the *memcached* benchmark we perform relatively well with an overhead of just 30% compared to *vanilla Linux*. The characteristics of Memcached is that it performs a lot of memory operations but relatively little system calls.

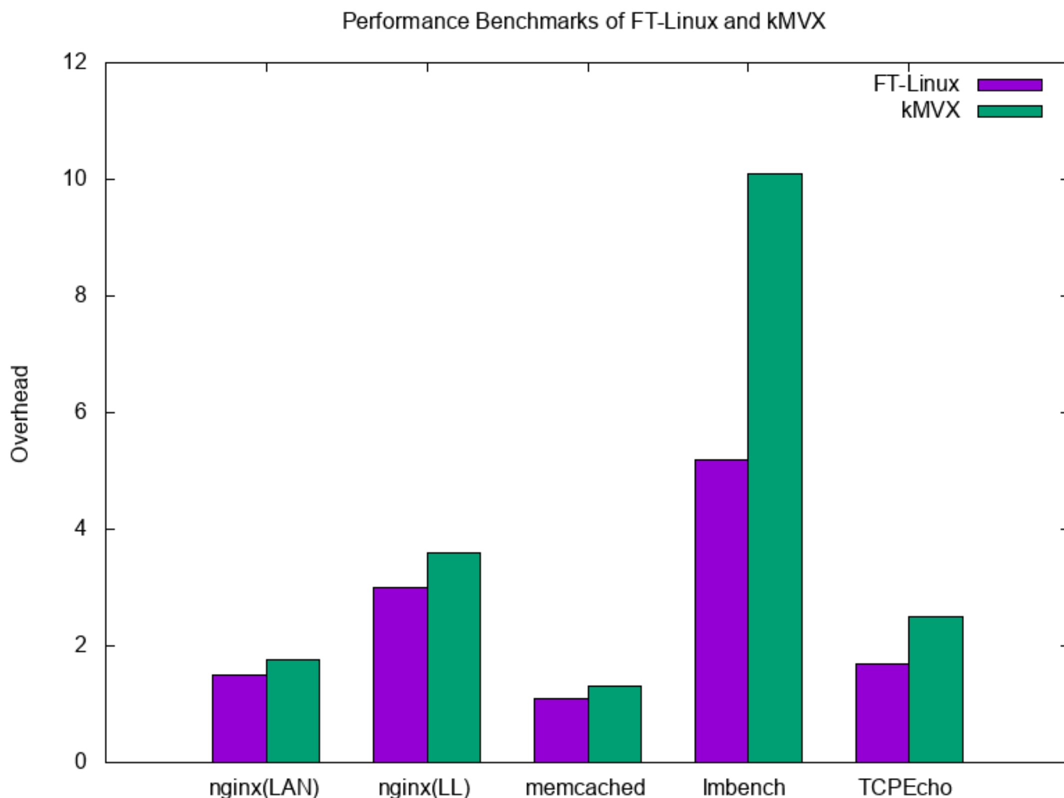

FIGURE 4.1: Macro benchmark results using full checks on all system calls.

As can be seen in the Figure 4.1 the network performance (i.e., the *nginx* benchmark) of FT-Linux in our measurements is much worse than presented in [47]. One reason for this may be that in the FT-Linux setup, the authors use a 32-core machine, while we only

have 2 cores. FT-Linux utilizes a number of work-queues for network synchronization equal to the number of cores, speeding up network replication significantly when running on systems with a higher core count. Furthermore, they only utilize a 1 GB/s connection which hardly puts the system under stress. We use a link-local connection, as to not limit the benchmark by the network bottleneck.

Micro benchmarks

In our micro benchmarks we get a more detailed overview of where our performance overhead originates. As stated earlier, our micro benchmarks include three different system calls that should show different characteristics. As seen in Figure 4.2, we incur more overhead per system call than FT-Linux.

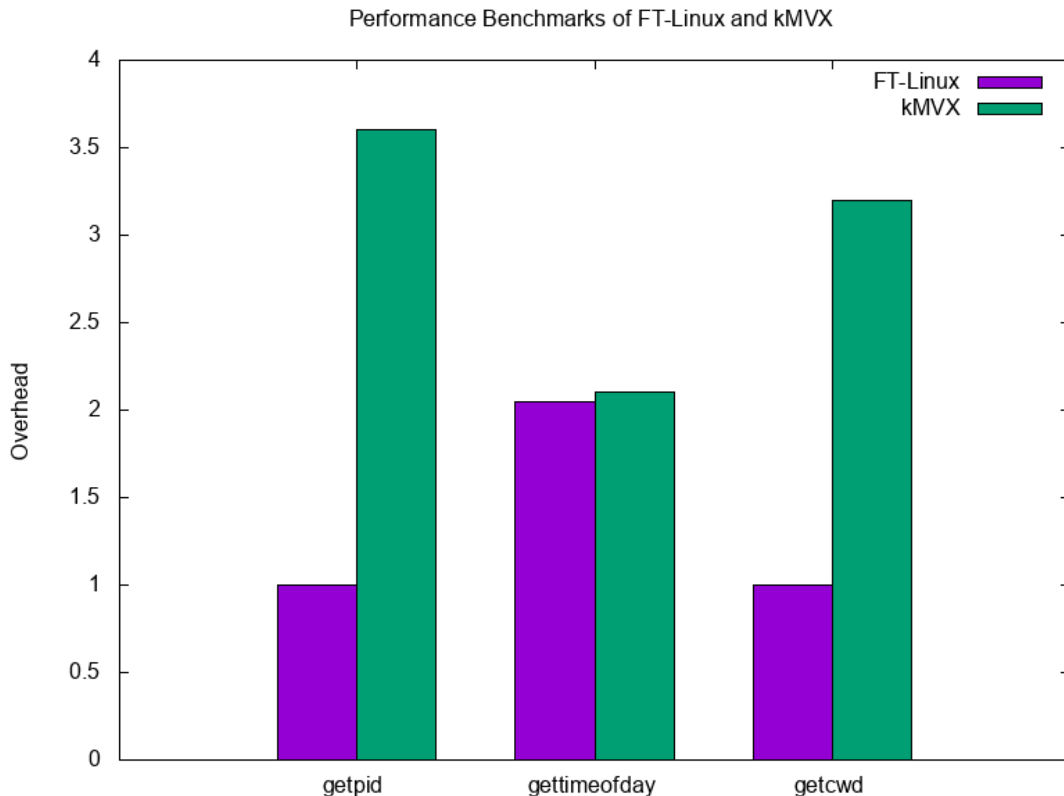

FIGURE 4.2: Micro benchmark results using full checks on all system calls

In particular `getpid` is much more costly (taking three times as long). This phenomenon is easily explained by the fact that FT-Linux does not need to perform any inter-kernel communication for the `getpid` system call. In kMVX, however, we need to send the return value — copied through `copy_to_user` — to the replica.

The `gettimeofday` system call incurs a relatively low overhead compared to `getpid`. The reason being, that, `gettimeofday` is one of the replicated system calls. FT-Linux already needs to communicate the result of this system call to the replicas, thus our additional message to the replicas does not add as much relative overhead as in the case of `getpid`. In hindsight, we could turn off the *barrier synchronization* for replicated system calls, as the values will never differ anyhow — unless, of-course there is an exploit in the modified system calls.

In the final system call micro benchmark, we repeatedly call `getcwd`. Note that the string length of the working directory from which the binary is called is 36 characters. We observe that the overhead of `getcwd` is relatively low compared to `getpid`, despite it needing to send more information to the replica kernel. This can be explained by the fact that message sizes in Popcorn Linux are cache aligned. If a messages total size goes over the statically defined 128 bytes, the message will be split into chunks. As our data (36 bytes) together with the Popcorn Message header size (5 bytes) fits into the 128 bytes available for one message, little additional overhead is incurred for the size of the buffer.

Additionally, we have a look at how the *buffer size* of a barrier buffer affects the performance. In Figure 4.3, we plot differently sized calls to read on an anonymously memory-mapped file. We notice, not completely unexpectedly, that the per-call cost of the system call grows as the buffer size increases. We see a jump in cost at the 256 byte size point. At this point the message has to be split into multiple parts when being sent to the other kernel.

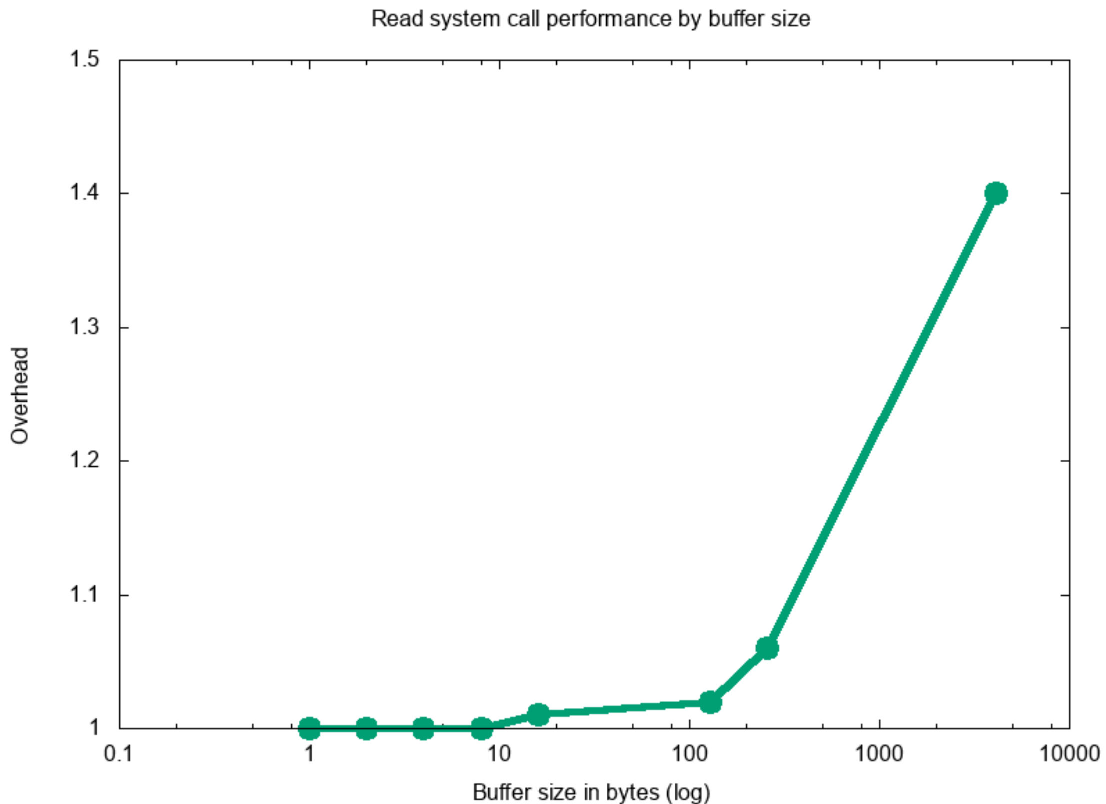

FIGURE 4.3: Buffer size benchmark. Micro benchmark of the `read` system call.

During development, we noticed that the replica kernel was always behind the primary kernel. This may be due to 1) worse performance in the memory allocator (the type-safe `kmalloc` is slower), 2) the fact that it has to wait for the master on every `copy_to_user` and 3) the fact that the process starts running earlier on the master than on the replica. We micro-benchmark our barriers to find out how much the variants spend waiting on each other. In Table 4.1 the time spent waiting on the other variant is presented. To eliminate the fact that the allocator is slower in the replica, we re-run the same benchmark using the default slab allocator. We notice a slight, but negligible, improvement in performance. Nevertheless, a majority of the master's time is spent waiting for the replica. Note that in our results, the replica never spends time waiting for the master.

TABLE 4.1: Micro benchmark results of synchronization overhead.

	type-safe kmalloc	*default kmalloc*
fraction spent waiting in master	0.75	0.71
fraction spent waiting in replica	0.0	0.0

Link-local full saturation test on faster hardware

When running our benchmarks on a different, more powerful machine, the system is less saturated. We, thus run network benchmarks using link-local networking. In this setup the baseline performance of Linux is much higher. This causes the base performance of FT-Linux to become much worse. Since kMVX is based on FT-Linux, our performance also tanks. Below we present a subset of the previously presented benchmarks, but run link-local on an i7. As previously noted, these results should be taken with a grain of salt, as FT-Linux was developed specifically to run on an AMD Opteron CPU, with different characteristics from the i7 we use in this setup.

Table 4.2: Overhead of benchmarks on i7 setup

	nginx	*gettimeofday*
FT-Linux	5.3x	3.2x
kMVX	6.7x	3.8x

4.3 Security Analysis

Using kMVX we can detect a number of different vulnerability classes. We now demonstrate using a few real-world kernel exploits, as well as artificial scenarios how our defenses successfully mitigate a large part of possible of exploits. First we observe that due to our variation techniques we can either detect exploitation attempts *deterministically* or *probabilistically*. The *deterministically* mitigated exploits are ones that make use of absolute addresses. For example, exploits that try to circumvent KASLR by leaking a kernel pointer, are deterministically prevented, as the set of valid pointers are disjoint between the variants. All vulnerabilities that rely on *absolute kernel addresses* are mitigated. The *probabilistically* mitigated exploits are the ones that make use of *relative addressing*. Our mitigation techniques prevent these kinds of exploits on a best-effort basis. For example, heap-based exploits may leak some kernel heap information. If the leaked memory area contains a kernel memory pointer it is detected. However, if the are contains sensitive information that is same across the variants, we cannot detect that the information has been leaked. However, since we use different heap allocators in the variants, such heap-based attack will most likely leak different data across variants, allowing us to detect the exploit. We now describe in detail how we detect some possible exploitation of the kernel.

Artificial vulnerability

We demonstrate, using a artificially created kernel vulnerability that our approach works. We modify the `mmap` system call, such that when we map data of a certain size with specific flags, some non-zeroed data from the kernel is leaked to user space. We exploit this vulnerability using a program that scans the mapped memory for non-null values, and prints the first non-null pointer value to the standard output. We detect the exploitation of the vulnerability in the following way:

1. The attacker maps memory using the malicious flags. This succeeds as we do not check the return value of `mmap` as the returned virtual address differs between replicas.
2. The attacker scans the memory, possibly finding a pointer in the leaked memory. Note that due to our variant generation strategy, the valid kernel pointers in the replicas are necessarily different.
3. The attacker tries to leak this pointer through the `write` system call.
4. `write` calls `copy_from_user`, which forces our variants to do a hard synchronization.
5. The buffers in `copy_from_user` are compared. We detect a divergence in one variant and, thus, the exploit is prevented.

Real-world vulnerabilities

Besides mitigating the artificial exploit we previously explained, we have also tested our system with some real-world exploits.

- **CVE-2013-7264**. "The l2tp ip recvmsg function in `net/l2tp/l2tp_ip.c` in the Linux kernel before 3.12.4 updates a certain length value before ensuring that an associated data structure has been initialized, which allows local users to obtain sensitive information from kernel stack memory via a (1) `recvfrom`, (2) `recvmmsg`, or (3) `recvmsg` system call" [52].

  We detect exploitation of this vulnerability since the sensitive information leaked from the kernel may be different due to out stack-layout variation strategies. Note, that we cannot deterministically detect this exploit, as the stack variation may not be enough and the leaked content may be the same across replicas (e.g., cryptographic keys). However, if the leaked content contains a leaked pointer, we will deterministically detect it.

- **CVE-2016-4569**. As discussed in Section 1.1.2, this vulnerability leaks information from the kernel stack due to uninitialized data in the `ioctl` system call. As with CVE-2013-7264, we detect this exploit *optimistically*.

Side-channel exploit detection

We have shown that we can detect leaking of sensitive information from the kernel obtained using *uninitialized data bugs* from the kernel. While information leaks can be harmless by themselves, the information gained from such an exploit may be used in further stages of an attack against the kernel. As mentioned in Section 1.3, hardware-based side-channel exploits have become increasingly common. In our design, we have incorporated a monitor which checks for possible side-channel attacks. If it detects that a process show unusual activity, our monitor will be activated for that process.

Suppose an attacker exploits a Javascript-based cache attack in the web-browser [41] to break KASLR, to further exploit a kernel privilege escalation vulnerability (e.g., using [53]) Our *pessimistic* abnormal activity detector would flag this event as suspicious, since the process running the JavaScript code would have a suspicious ratio of *cache misses* to *actual cache accesses*, thus enabling kMVX monitoring. Due to our design, with different kernels run on different cores having their own memory mapping. Since our variants have disjoint virtual memory areas, the leaked area locations would be different across variants. If the leaked absolute addresses were used in a later exploitation stage, we would detect it. Also note, that the obtained information is not leaked yet to the outside world; it is still contained in user-space, which is considered a closed system. If an attacker would send the leaked kernel addresses over the network, kMVX would detect the deviation in to other variant when calling `copy_from_user`. Since we enforce a hard barrier at that sink, we will prevent the information from leaving the system.

As mentioned in Section 3.3.3, the question arises whether we can actually detect a side-channel timing attack **before** it is able to leak information. We set up a test environment to demonstrate that we are able to **prevent** confidential information from leaking the system. By using a *pre-fetch* cache attack, as described by Gruss in [39], we circumvent KASLR by leaking mapped memory areas. In our created test program we leak the obtained mappings by printing them to the standard output. Before an attack is detected, we do not enforce a hard synchronization on `copy_from_user`. When we detect a possible cache-attack, we immediately enable `copy_from_user`-synchronization. In our setup we run the described attack 100 times, and measure whether kMVX enables `copy_from_user` synchronization before the attack manages to print any obtained values to the standard output.

TABLE 4.3: **Side-channel prevention**. Confusion matrix describing whether we detect timing attacks before information leaves the system.

Detected	Yes	No
Leaked		
Yes	0.19	0.11
No	0.70	0

As can be seen in Table 4.3, our current static model (flagging processes with a miss-hit-ratio of over 0.7) is capable of detecting 89% of the attacks.

While this is a single, heavily specific scenario, we argue that this demonstrates that, depending on the detection mechanism used, kMVX can be **successfully applied to preventing information leaking** the system in such. Since our current approach requires at least 60 samples, 9% of the leaking attempts succeed. By utilizing *smarter detection mechanisms* the number of samples required (and thus the latency of detecting an attack) is reduced significantly. For example, in [43] Chiappetta & al. describe a machine-learning approach that requires just 21 samples to detect attacks with an *F-Score* of 1.

Chapter 5

Discussion

5.1 Why kMVX?

In the previous chapters, we have discussed common threats against modern Operating Systems. We have discussed the design of a comprehensive kernel exploit mitigation system, based on Multi-Variant Execution in the kernel. We have shown using a prototype implementation, that, while the overhead of using such a comprehensive defense mechanism continuously may be a bit too costly at the moment, we can enable the defenses as needed. We have shown that by enabling the defenses when a possible exploitation attempt is detected, we can prevent a large number of exploits. As far as we know, this is the first system that can effectively defend a operating system against side-channel timing attacks breaking Kernel Address-Space Layout Randomization without *false-positives*.

We have demonstrated our performance in various benchmarks, putting stress on different kinds of systems in the kernel. We have demonstrated the use of kMVX in a server setting with *Nginx* and *Memcached*.

5.2 Drawbacks

There are some drawbacks with our prototype implementation. First, it relies on Popcorn Linux/ FT-Linux, which is based on a older version of the Linux kernel (more precisely 3.2.14 released in march 2012). Second, the focus in FT-Linux is not on security, thus certain implementation aspects may be inappropriate from a security perspective.

5.2.1 Need for more kernel variants

In this thesis, we have identified a number of possible variation techniques for kernels. Despite the different variation techniques, we do not yet have an easy-to-apply variation technique In our prototype, much of the variant generation is manual. Some compiler-based variation techniques have been utilized, and our custom memory allocator can easily be modified to use a different hashing function to introduce more variation. As noted, automatic variant generation can have unintended consequences (e.g., performance), that are much more noticeable than in user-space. A limitation in our benchmarking is that we have only used two replicated kernels. Due to time-constraints and hardware constraints, we were not able to perform benchmarks on a setup with more variants.

5.2.2 Single point of failure

Since we are preventing kernel exploits and our system is implemented in kernel-space, we have to consider possible attacks against kMVX that mitigate our defense mechanisms. In the current design, there is one major issue. Suppose an attacker makes use of an exploit that affects the master replica, and as such gains control of the master replica. Our whole defense strategy is side-stepped by the fact that the master replica is the one that single-handedly decides whether an exploit has been detected. We propose modifying our system, such that the different variants broadcast their values at the barrier to all other variants. This makes it possible to detect if there is a divergence in the execution that the master does not acknowledge. By a consensus algorithm, the variants can elect a new master. Naturally, the question arises: how do we decide which variant tells the truth? Suppose a compromised node sends out different values to the nodes when broadcast its values to influence the leader election? One solution is to apply a Byzantine fault tolerance algorithm [54]. This would add some overhead as these kinds of distributed algorithms are expensive. However, since this fault detection would only kick in when something strange has happened anyway, the overhead incurred would be well worth it to prevent further exploiting of the system.

5.2.3 Replay vs. multi-variant

Some possible attack vectors are missed in our implementation due to replayed system calls that are only executed on the master. Traditionally, the network stack — due to its complexity — has been a large attack vector. Since we replay most of the system calls to and from the network stack, we will miss a number of vulnerabilities that can be

exploited in it. Ideally, we would like to run more of the network stack across variants. In the next section we propose some ideas to fix these issues.

5.3 Future extensions

In the previous chapters we have pointed out some problems with the current implementation that could be improved. Here we discuss a number of possible future improvements.

5.3.1 Network-performance improvement

One of the great bottlenecks in the current implementation is due to the required replaying of network system calls. Furthermore, replaying these system calls rather than actually doing state replication on the replicas decreases the attack area that we can detect. We propose to utilize PCI-E 2.0 dual cast [55] functionality to set multiple *egress ports* for the NIC — one for each kernel— such that the network stack is totally replicated, this eliminates the need for the replicas to wait for the master when receiving network packets.

5.3.2 vDSO

Popcorn Linux support user-space message-passing through user-space shared memory. By implementing a custom variant of the vDSO, we could eliminate the need to trap into ring-0 when executing system calls like `gettimeofday` and `getpid`.

5.3.3 Remaining sources of divergence

In our implementation we have ignored some sources of divergence, as these have not been noticeable in our test-cases and benchmarks. For example, `rdtsc` could return different values in the different variants. This source of divergence can be disabled by setting the `TSD` flag in the `cr4` register of all replicated processes. As `rdtsc` is a hardware feature, it is hard to ensure that the replicas return a consistent value for this call between replicas. For the sake of completeness, we implement replaying of the `rdtsc` instruction by catching all calls to the instruction for processes in the replicated namespace. Disabling `rdtsc` (by setting the `TSD` bit in the `CR4` register) will generate an *illegal instruction* trap, allowing us to intercept all such calls. After catching a call to `rdtsc`, we read the Time Stamp Counter in kernel-mode, and finally replay the return value from

the master. Naturally this slows down `rdtsc` remarkably. Note that for our benchmarks we enable the `rdtsc` instruction to make our (micro) benchmarking possible. Similarly, reading from `/dev/random` may return different values in the different variants. We propose replacing `/dev/random` by a per-process *random number generator* with the same seed used on all replicas.

5.3.4 File system

Another problem is the file-system. In our prototype, we ignore the problem of multiple divergent file-systems by ignoring difference in return values of calls such as `fstat`.For our benchmarks we have made sure that the file-systems are largely identical. Another problem is that the kernels return different *file descriptor ids* across variants since these are generate globally in the kernel. We can modify the existing id generation system, like we have with the process id, to generate file descriptors that are consistent across the replicated processes. For a production-ready system such divergence must be eliminated. On solution is to make use of replaying for file-system calls. However, as we stated *this reduces our detection surface* significantly.

We propose adding an additional layer on top of the file-system, residing in memory. When reading from the file-system, we encounter a *sink*. As sinks are not an issue for information leaks we do need to enforce a hard synchronization. We could let one variant read the file from the actual disk, placing it in an *emulated* in-memory filesystem from which the other variants can read. Writing to the disk is harder, as this is a *sink*. If one variant writes before others have reached that point, sensitive information can leave the system. We propose to solve this by a *copy-on-write shadow filesystem*. When one replica writes to the filesystem, its contents are placed in a barrier-specific *bucket* in the emulated *in-memory* file-system. Furthermore, each replica stores a hash of the written content in the bucket. When all replicas have written to the bucket, the master takes care of comparing the hashes of all variants. If all the hashes match, the content is written to the actual disk.

5.3.5 Inter-kernel communication

Popcorn Linux currently utilizes a *multi-writer single-reader* queue for its ringbuffer. For kMVX a *single-writer single-reader* would suffice. Applying a different access model for the ringbuffer would allow us to utilize a **lock-free queue**, as presented by Herlihy in [56], eliminating the need for costly *locks*.

Besides the ringbuffer locking mechanism, the way inter-kernel messaging is handled in Popcorn Linux is sub-optimal for kMVX's *small-but-frequent* messaging characteristics. When a message is sent from one kernel to another, the message is added to the ringbuffer, after which an interrupt is sent to the receiving end. As we require a large number of inter-core messages, this *interrupt storm* may decrease the performance significantly. We propose to only send an interrupt if a *certain number of messages* have been buffered with a fallback of sending an interrupt *once every (preset) interval* in case there are not enough messages to fulfill our first criterion. By limiting the number of interrupts being sent per inter-kernel message, the *interrupt storm* phenomenon should be avoided. Another better performing alternative would be to utilize hardware functionality such as `MWAIT` to implement the queue polling, as discussed by Hruby & al. in [57], eliminating the need for busy waiting.

5.4 Conclusion

In this thesis we have discussed the design of a kernel-level multi-variant execution system – to our knowledge the first application of MVX to kernels — with the goal of protecting the kernel against exploits. For now we have limited the scope to local attacks, with a focus on information disclosure. We have shown an efficient, generic multi-kernel design that can be applied to other operating systems as well. We demonstrate our design by implementing a prototype based on the Popcorn Linux multikernel. Despite major modifications, we can still run normal Linux binaries on our system, making it easy to deploy into existing infrastructures.

Despite the delicate nature of Operating System Kernels, we have been able to apply multiple existing variation techniques to this area. Our comprehensive defenses can mitigate exploits, ranging from *memory corruptions* to *information leaks*. We implement a system detecting possible hardware-based side-channels timing attacks, with no *false-positives*, while also demonstrating that our design can prevent actual, real-world kernel exploits while having a reasonable overhead for most use-cases.

Bibliography

[1] Victor Van der Veen, Nitish Dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: the past, the present, and the future. *Research in Attacks, Intrusions, and Defenses*, pages 86–106, 2012.

[2] Arash Baratloo, Navjot Singh, Timothy K Tsai, et al. Transparent run-time defense against stack-smashing attacks. In *USENIX Annual Technical Conference, General Track*, pages 251–262, 2000.

[3] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[4] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, pages 51–66, 2009.

[5] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceedings of the 28th international conference on Software engineering*, pages 162–171. ACM, 2006.

[6] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[7] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 191–205. IEEE, 2013.

[8] Steve McConnell. Code complete: a practical handbook of software construction (redmond, wa, 1993.

[9] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[10] Mark K Joseph and Algirdas Avizienis. A fault tolerance approach to computer viruses. In *Security and Privacy, 1988. Proceedings., 1988 IEEE Symposium on*, pages 52–58. IEEE, 1988.

[11] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*, pages 431–442. IEEE, 2016.

[12] BH SHELDON. Popcorn linux: enabling efficient inter-core communication in a linux-based multikernel operating system. *Master's thesis, Virginia Polytechnic Institute and State University*, 2013.

[13] Linus Torvalds. Linux: a portable operating system. *Master's thesis, University of Helsinki, dept. of Computing Science*, 1997.

[14] Cve details linux kernel, aug 2017. URL `https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33`.

[15] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 5. ACM, 2011.

[16] Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique, 2005.

[17] 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*, pages 552–561. ACM, 2007.

[18] Aditi Gupta, Sam Kerr, Michael S Kirkpatrick, and Elisa Bertino. Marlin: A fine grained randomization approach to defend against rop attacks. In *Network and System Security*, pages 293–306. Springer, 2013.

[19] Michael Backes and Stefan Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security Symposium*, 2014.

[20] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*, pages 475–490, 2012.

[21] Fermin J Serna. The info leak era on software exploitation. *Black Hat USA*, 2012.

[22] 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.

[23] CVE-2016-4569. Available from MITRE, CVE-ID CVE-2016-4569., May 10 2016. URL `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4569`.

[24] CVE-2014-1444. Available from MITRE, CVE-ID CVE-2014-1444., January 18 2016. URL `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1444`.

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

[26] Babak Salamat, Andreas Gal, Todd Jackson, Karthikeyan Manivannan, Gregor Wagner, and Michael Franz. Multi-variant program execution: Using multi-core systems to defuse buffer-overflow vulnerabilities. In *Complex, Intelligent and Software Intensive Systems, 2008. CISIS 2008. International Conference on*, pages 843–848. IEEE, 2008.

[27] Todd Jackson, Christian Wimmer, and Michael Franz. Multi-variant program execution for vulnerability detection and analysis. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, page 38. ACM, 2010.

[28] Stijn Volckaert, Bart Coppens, and Bjorn De Sutter. Cloning your gadgets: Complete rop attack immunity with multi-variant execution. *IEEE Transactions on Dependable and Secure Computing*, 13(4):437–450, 2016.

[29] 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.

[30] Todd Jackson, Babak Salamat, Gregor Wagner, Christian Wimmer, and Michael Franz. On the effectiveness of multi-variant program execution for vulnerability detection and prevention. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, page 7. ACM, 2010.

[31] Babak Salamat, Andreas Gal, and Michael Franz. 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.

[32] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280. ACM, 2003.

[33] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[34] Zhaohui Liang, Bin Liang, and Lupin Li. A system call randomization based method for countering code-injection attacks. *International Journal of Information Technology and Computer Science (IJITCS)*, 1(1):1, 2009.

[35] 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.

[36] 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.

[37] Bryan Buck and Jeffrey K Hollingsworth. An api for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[38] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 380–392. ACM, 2016.

[39] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 368–379. ACM, 2016.

[40] Michael Schwarz and Anders Fogh. Drama: How your dram becomes a security problem. *Black Hat Europe*, 2016.

[41] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Christiano Giuffrida. Aslr on the line: Practical cache attacks on the mmu. *NDSS (Feb. 2017)*, 2017.

[42] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware

detection with performance counters. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 559–570. ACM, 2013.

[43] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162–1174, 2016.

[44] Mathias Payer. Hexpads: a platform to detect stealth attacks. In *International Symposium on Engineering Secure Software and Systems*, pages 138–154. Springer, 2016.

[45] Sape J. Mullender, Guido Van Rossum, AS Tananbaum, Robbert Van Renesse, and Hans Van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, 1990.

[46] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM, 2009.

[47] Barbalace A. Losa G. Transparent fault-tolerance using intra-machine full software stack replication. *The 37th IEEE International Conference on Distributed Computing Systems (ICDCS 2017)*, 2017.

[48] Keisuke Okamura and Yoshihiro Oyama. Load-based covert channels between xen virtual machines. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 173–180. ACM, 2010.

[49] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, page 27, 2008.

[50] Dmitrii Zagorodnov, Keith Marzullo, Lorenzo Alvisi, and Thomas C Bressoud. Engineering fault-tolerant tcp/ip servers using ft-tcp. In *DSN*, pages 393–402, 2003.

[51] memaslap - load testing and benchmarking a server, aug 2017. URL `http://docs.libmemcached.org/bin/memaslap.html`.

[52] CVE-2013-7264. Available from MITRE, CVE-ID CVE-2013-7264., January 6 2013. URL `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-7264`.

[53] CVE-014-3153. Available from MITRE, CVE-ID CVE-014-3153., May 3 2014. URL `https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-3153`.

[54] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[55] Jayakrishna Guddeti and Luke Chang. Dual casting pcie inbound writes to memory and peer devices, November 17 2015. US Patent 9,189,441.

[56] Maurice P Herlihy and Jeannette M Wing. Axioms for concurrent objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 13–26. ACM, 1987.

[57] Tomas Hruby, Dirk Vogt, Herbert Bos, and Andrew S Tanenbaum. Keep net working-on a dependable and fast networking stack. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.