UNIVERSITY OF CALIFORNIA,
IRVINE

Building the Next Generation of Security Focused NVX Systems: Overcoming Limitations of N-Variant Execution

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Alexios Voulimeneas

Dissertation Committee:
Professor Michael Franz, Chair
Prof. Alex Nicolau
Prof. Qi Alfred Chen
Prof. Joshua Garcia
Prof. Stijn Volckaert

2020

DEDICATION

This dissertation is dedicated to *my* people. People, who were by my side when I was the best and the worst version of myself, including friends, my family and Lydia. I would like to mention my friend Dimitris and my twin sister Christina seperately. You are my heroes and your misfortune made me to value life more, enjoy the moment and realize that you only live once. Life is a gift and we should treat it like that!

–Alexios Voulimeneas, April 2020

"Because you relied on someone else's script, you were wrong. I won't live my life by another person's script. Not by Hamlet or The Tempest. I don't know how many years in the future it will be... but I'll write and act my own ending."

–Kyo Shirodaira, Blast of Tempest (Mahiro Fuwa)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express my heartfelt thanks to my advisor, Prof. Michael Franz. Over the past five years, PhD was a roller coaster for me and he has been an incredible mentor, providing (more than enough) guidance and encouragement throughout my graduate studies at UCI. I sincerely believe that working with Michael made me a better researcher and a better person.

I would like to thank my past and current postdocs, Prof. Stijn Volckaert (previously postdoc at UCI), Dr. Per Larsen, Dr. Yeoul Na, Dr. David Gens and Dr. Adrian Dabrowski for their contributions, ideas and advices. I am also grateful to lab alumni, Andrei Homescu, Stephen Crane, Brian Belleville, Mohaned Qunaibit and Julian Lettner for their invaluable support and willingness to share their expertise and experiences. In addition, I would like to thank my labmates for being great colleagues and for making graduate school much more fun. Prabhu, Joseph, Dokyung, Min, Mitch, Fabian, Chinmay and Paul–it has been a pleasure working and hanging out with you. Taemin and Anil, I would like to thank you seperately; we are the three *musketeers* of UCI and I consider you part of my UCI family.

Furthermore, I would like to express my deepest gratitude to several other PhD students, researchers, professors and practicioners. Prof. George Xylomenos, Prof. Stavros Toumpis, Prof. Vasileios Kemerlis, Charilaos Stais, Christos Tsilopoulos, Esmerald Aliaj, Georgios Detorakis, Chris Mariolas, Lefteris Kokoris-Kogias, Ioannis Gasparis, Marios Pomonis, Nikolaos Ignatiadis, Kostis Kaffes, Faidra Monachou, Dimitrios Belivanis, Vasileios Nakos, Vaggos Chatziafratis, Andrew Hansen, Laceé Pappas, Evelia Salinas, Harry Fokas, Eric Gonzalez, Travis Bartley, Armaan Saini, James Yamagata and Abdulahman Alsaudi–thank you all from the bottom of my heart.

Finally, I would like to give special thanks to my committee members, Prof. Alex Nicolau, Prof. Qi Alfred Chen, Prof. Joshua Garcia and Prof. Stijn Volckaert for their time and consideration.

Portions of this dissertation have been previously published in conference proceedings. To my coauthors on these projects, thank you for your contributions on these publications.

Portions of Chapter 2 and Chapter 3 are reprinted, with permission, from Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, Michael Franz. *Secure and Efficient Application Monitoring and Replication.* In Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC 16).

Portions of Chapters 3, 4, 5, 6 and 7 are reprinted, with permission, from Alexios Voulimeneas, Dokyung Song, Fabian Parzefall, Yeoul Na, Per Larsen, Michael Franz and Stijn Volckaert. *Distributed Heterogeneous N-Variant Execution.* To appear in the 2020 International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2020).

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124, FA8750-15-C-0085, FA8750-

10-C-0237, and FA8750-16-C-0260, by the United States Office of Naval Research (ONR) under contract N00014-17-1-2782, by the National Science Foundation under award numbers CNS-1513837, and CNS-161921 as well as gifts from Mozilla, Oracle, and Qualcomm.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA) or its Contracting Agents, the Office of Naval Research (ONR) or its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.

VITA

Alexios Voulimeneas

EDUCATION

Doctor of Philosophy in Computer Science — **2020**
University of California, Irvine — *Irvine, California*

Master of Science in Computer Science — **2017**
University of California, Irvine — *Irvine, California*

Degree in Informatics, 4-year curriculum — **2012**
Athens University of Economics and Business (AUEB)
Department of Informatics — *Athens, Greece*

RESEARCH EXPERIENCE

Graduate Student Researcher — **2015–2020**
University of California, Irvine — *Irvine, California*

Visiting Scholar — **Fall 2019**
KU Leuven — *Ghent, Belgium*

Undergraduate and Graduate Researcher — **2011–2014**
Mobile Multimedia Laboratory/AUEB — *Athens, Greece*

WORK EXPERIENCE

Software Engineering Intern — **Summer 2019**
Apple Inc — *Cupertino, California*

Research Assistant Intern — **Summer 2017**
Oracle Labs — *Belmont, California*

Hellenic Army (Mandatory Military Service) — **2014–2015**
Research & Informatics Directorate — *Greece*

Software Engineering Intern — **Summer 2012**
NCSR Demokritos — *Athens, Greece*

TEACHING EXPERIENCE

Teaching Assistant **Winter 2017**
ICS 6B: Boolean Algebra & Logic
University of California, Irvine *Irvine, California*

Teaching Assistant **Summer 2010**
Computational Mathematics
Athens University of Economics and Business (AUEB) *Athens, Greece*

PUBLICATIONS

Distributed Heterogeneous N-Variant Execution **DIMVA 2020**
A. Voulimeneas, D. Song, F. Parzefall, Y. Na, P. Larsen, M. Franz and S. Volckaert
International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (To appear)

Secure and Efficient Application Monitoring and Replication **ATC 2016**
S. Volckaert, B. Coppens, **A. Voulimeneas**, A. Homescu, P. Larsen, B. De Sutter and M. Franz
USENIX Annual Technical Conference

A Reliable Multicast Transport Protocol for Information-Centric Networks **JNCA 2014**
C. Stais, G. Xylomenos and **A. Voulimeneas**
Journal of Network and Computer Applications

Towards an Error Control Scheme for a Publish/Subscribe Network **ICC 2013**
C. Stais, **A. Voulimeneas** and G. Xylomenos
International Conference on Communications

SERVICE

IEEE S&P Student PC **2018**

DISTINCTIONS

ACM CCS student travel award	**2016**
IEEE S&P student travel award	**2016**
ICS Dean's Award University California, Irvine	**2015**
Scholarship for graduate studies at EPFL Latsis Foundation	**2013**
Scholarship for graduate studies at EPFL Foundation for Education and European Culture	**2013**

ABSTRACT OF THE DISSERTATION

Building the Next Generation of Security Focused NVX Systems:
Overcoming Limitations of N-Variant Execution

By

Alexios Voulimeneas

Doctor of Philosophy in Computer Science

University of California, Irvine, 2020

Professor Michael Franz, Chair

N-Variant Execution (NVX) systems utilize artificial diversity techniques to enhance software security. The general idea is to run multiple different variants of the same program alongside each other while monitoring their run-time behavior. If a malicious input causes the execution paths of the diversified variants to diverge, the monitor can detect divergences, e.g. at the system call level, and take defensive action.

Several NVX systems have been proposed over the last two decades, providing different security/performance characteristics. In general, security-oriented NVX systems greatly degrade performance, while high performance NVX systems have two disadvantages; they significantly increase the size of the Trusted Computing Base (TCB) or sacrifice security. In this dissertation, we want to investigate if it is possible to build an alternative NVX design that unifies the strengths of existing approaches.

Security-oriented NVX systems are considered *strong* defenses that protect against a variety of attacks. However, a subset of modern attacks have the potential to bypass existing NVX systems. We identify *limited* available diversity as the main reason for that. Existing NVX systems execute diversified program variants on a single host. This means that the level of inter-variant diversity will be limited to what a single platform can offer.

The main focus of this dissertation is to investigate the possibility of building the first distributed heterogeneous NVX system that executes program variants across multiple heterogeneous hosts. This approach can increase the level of internal diversity between the simultaneously running variants that can be supported, encompassing different instruction sets, endianness, calling conventions, system call interfaces, and differences in hardware security features. We expect that new challenges will arise from the distributed and heterogeneous nature of this design, however we believe that we will be able to provide sufficient solutions.

"Empty your memory,

with a free()...

like a pointer!

If you cast a point to an integer,

it becomes the integer,

if you cast a pointer to a struct,

it becomes the struct...

The pointer can crash...,

and can Overflow...

Be a pointer my friend..."

–Dennis Ritchie

Chapter 1

Introduction

Software vulnerabilities have great implications, since software is used in all aspects of our lives, including the health and financial sectors. Specifically for systems and embedded programming, the root cause is the use of unsafe languages such as C and C++. C/C++ are highly efficient languages that provide direct control over hardware resources and memory management. However, applications implemented in C/C++ are prone to memory errors. Several techniques have been proposed by practitioners and researchers to defend against classes of these attacks, but nonetheless, attackers always find ways to bypass them. While programming languages such as Rust, provide full or partial memory safety, there are millions lines of legacy code written in C/C++ that we still need to protect.

1.1 Overview

Memory errors are a continuous source of software vulnerabilities for C and C++ programs. Attackers and defenders are engaged in an arms race in which the latter keep developing sophisticated defenses while their adversaries create new exploits that bypass these

defenses [69]. At present, adversaries rely on intimate knowledge of the target environment (such as details about the victim application, the target operating system and the instruction set architecture of the host) to mount code-reuse [65, 13, 66] or data-oriented attacks [15, 36, 35] that allow them to take control of the target or leak its sensitive data. While memory safety techniques protect against these threats, many of these techniques have not seen widespread deployment due to performance [51, 52] and compatibility problems [67].

Instead, defenders resort to mitigations that have a more reasonable performance impact, *e.g.*, control-flow integrity (CFI) techniques [5, 49, 11], automated software diversity techniques [43, 31, 32], or a combination thereof. However, both classes of defenses have a history of known weaknesses: CFI defenses often still leave some leeway to mount control-flow hijacking attacks [17, 21, 72]. Software diversity techniques have been bypassed using brute-forcing and information leakage attacks, including attacks enabled by micro-architectural side channels [66, 9, 28, 37].

1.2 N-Variant Execution

N-Variant eXecution (NVX) systems amplify the effectiveness of software diversity techniques and increase resilience [8, 10, 18, 61, 79, 47, 33, 39, 34, 42, 40, 76, 78, 82, 46, 22, 84]. An NVX system runs multiple diversified variants of the same program in parallel on the same inputs while monitoring the variants' behavior for divergences. If a malicious input causes the execution paths of the diversified variants to diverge, the monitor can detect divergences at the system call level and take defensive action. With the right selection of diversity techniques, NVX can make successful exploitation substantially harder (and, in some cases, even provably impossible [76]) as it forces adversaries to simultaneously compromise multiple program variants without causing observable changes in their behavior.

Several NVX systems have been proposed over the last two decades, providing different security/performance trade-offs. In general, security-oriented NVX systems come at the cost of high run-time overhead, while high performance NVX systems cannot provide adequate security and focus instead on reliability, *e.g.*, testing new patches. We present ReMon, a *new*, security-focused NVX system that is able to perform on par with reliability-oriented NVX systems, without increasing the TCB or sacrificing security.

Existing N-Variant systems have been successful in defending a variety of attacks. For example, we can completely eliminate all the attacks that use absolute addresses, by forcing code and data addresses to be unique in each variant [76, 18, 10]. However, existing NVX systems can be bypassed from modern attacks such as Position-Independent Return-Oriented Programming (PIROP) attacks [26] and certain Data-Oriented Programming (DOP) attacks [36], which build on knowledge of the program's internal geometry and/or data representation. We identify *limited* available diversity as the main reason for that.

We present and evaluate in high detail DMON, a *novel* NVX system that allows us to leverage the diversity that naturally exists across different platforms, thereby increasing resilience to memory exploits. DMON runs each program variant natively on its own dedicated machine and monitors divergent behavior between these distributed variants by monitoring them at the system call boundary using a reliable network channel. To bypass DMON, adversaries would need to develop exploits that work simultaneously against the two or more different ISAs and ABIs for which the program variants are compiled. DMON runs on commodity hardware with regular multi-core CPUs.

DMON faces performance challenges, due to its unique distributed design. We implemented several optimizations to improve its performance. In addition, we present, DMON++, a *new*, hybrid NVX system, based on DMON, that drastically reduces DMON's overhead even more.

1.3 Contributions and Structure of the Dissertation

This dissertation presents the following contributions, organized by chapter:

Chapter 2 In this chapter, we describe the high-level design principles and trade-offs of existing NVX systems. In addition, we present ReMon, a *new*, hybrid NVX system that induces low overhead without sacrificing security or increasing the Trusted Computing Base (TCB) significantly. By using selective monitoring of system calls, we are able to provide flexible security/performance characteristics according to our needs.

Chapter 3 In this chapter, we describe the reason that a subset of modern attacks are able to bypass the aforementioned NVX systems. Then we present DMON, a *novel* NVX system that raises the bar for these state-of-the-art attacks and discuss our design choices. DMON is the first system that combines ISA and ABI heterogeneity with N-Variant Execution. DMON distributes the execution of a set of variants over a heterogeneous set of physical machines.

Chapter 4 In this chapter, we briefly describe the causes of inconsistencies and false positive detections (or false alarms) one might encounter when running programs in a centralized NVX system, *e.g.*, shared memory support and asynchronous signal delivery. We also show DMON's limitations regarding the aforementioned problems and how we could adjust existing solutions with some additional engineering effort.

Furthermore, we extend the discussion of false positive detections for our distributed and heterogeneous setting. We show that our unique setting causes *new* false positives that have never been shown before in NVX systems. We also suggest solutions that tolerate or completely eliminate these false alarms without weakening the NVX system's security.

The core of our approach is a novel system call metadata transformation called Platform-Independent State Canonicalization (PISC), that translates system call states in different platforms into platform-independent states.

Chapter 5 In this chapter, we present ways to reduce the performance overheads associated with a distributed NVX system. Our results show that with the proposed performance optimizations, DMON performs on par (or better) with existing, cross-process, centralized NVX systems while providing stronger security guarantees.

Chapter 6 In this chapter, we evaluate DMON's security on several server applications and show that DMON makes code-reuse attacks substantially more difficult, and that it naturally provides a high degree of structure layout diversity which raises the bar for attacks that rely on consistent structure layout.

Chapter 7 In this chapter, we evaluate DMON's performance on microbenchmarks and complex applications. We show DMON's overhead, when different sets of our optimizations are enabled.

Chapter 8 In this chapter, we present DMON++, a *new*, hybrid, distributed NVX system, inspired from ReMon. DMON++ is implemented over DMON and our evaluation shows that it reduces drastically DMON's performance overhead.

Chapter 9 In this chapter, we discuss future directions, show related work and conclude the dissertation.

Chapter 10 This chapter discusses the contribution of the author of this dissertation in ReMon and DMON research projects.

Chapter 2 is based on the following conference article:

- S. Volckaert, Bart Coppens, **Alexios Voulimeneas**, Andrei Homescu, Per Larsen, Bjorn De Sutter and Michael Franz.
  Secure and Efficient Application Monitoring and Replication.
  In *USENIX Annual Technical Conference 2016*.

Chapters 3, 4, 5, 6 and 7 are closely based on the following article:

- **Alexios Voulimeneas**, Dokyung Song, Fabian Parzefall, Yeoul Na, Per Larsen, Michael Franz and Stijn Volckaert.
  Distributed Heterogeneous N-Variant Execution.
  To appear in *International Conference on Detection of Intrusions and Malware & Vulnerability Assessment 2020*.

Chapter 2

N-Variant Execution

N-Variant Execution systems (NVX) run multiple variants of the same program in lockstep. NVX systems provide the same input to all the variants and monitor their run-time behavior. The variants are constructed carefully in order to be functionally equivalent but to diverge under attacks. In recent years, several NVX systems have been proposed by academics that match the above description. However, authors have made different design choices that also come with security/performance trade-offs.

2.1 High-Level Monitor Design

The *core* of every NVX system is a monitor component. The two main properties of the monitor component are the monitoring granularity and where it is placed in the software stack. In this chapter, we describe different monitor designs and their trade-offs regarding security and performance.

2.1.1 Monitoring Granularity

We could monitor program variants at different granularities, from the machine instruction level, to system calls or just I/O operations. In practice, the vast majority of NVX systems monitor at the system call interface. NVX systems that monitor system calls, either monitor *all* system calls or separate them into *non-sensitive* and *sensitive* and only monitor the latter ones. Selective (or relaxed) monitoring uses different policies (with different security/performance characteristics), depending on the class of attacks that we want to defend against.

In modern operating systems there is a clear separation between user space and kernel space. Applications run at user space and low-level operations and hardware resources are *hidden* in the kernel space. In case, that a program wants to do something meaningful, it needs to use system calls, which is a well-predefined interface for communication between user space and kernel space. The same goes for attackers. The end goal of many attacks is to execute a sequence of `sys_mprotect` and `sys_execve` system calls.

Monitoring at finer granularity than system calls, *e.g.*, machine instructions, makes little sense. First of all, more fine-grained monitoring comes at the cost of higher performance overhead. Secondly, it limits the amount of available diversity between program variants. Finally, previous work showed that run-time divergences under attacks are visible at system call interface by constructing variants carefully [18, 61].

2.1.2 System Calls and I/O Replication

NVX systems monitor behavior and replicate I/O at the system call interface. This design lets the system monitor all behavior that can potentially affect the integrity of the OS or

other processes, as well as all communication between the variants and external entities[1]. Both monitoring and replication must be transparent to the program variants and to the end-user. In other words, neither the variants, nor any external observer should be able to notice any differences (apart from timing) between native execution of a single variant and NVX of multiple variants. To provide this guarantee, most NVX systems designate one variant as the leader, while the others become followers (see Figure 2.1). Whenever the variants attempt an I/O operation, the NVX systems ensure that only the leader variant actually completes the operation, while the followers skip the operation and wait until they receive the I/O results from the monitor; we call this procedure "replication". Replication is an essential mechanism of NVX systems since it ensures user-transparency (*e.g.,* when two variants want to write to a file, only the leader variant "actually" writes to it) and provides identical system call results to both variants (system call results are used as inputs to subsequent operations). Both NVX systems presented in this dissertation [78, 80] use this leader/follower model.

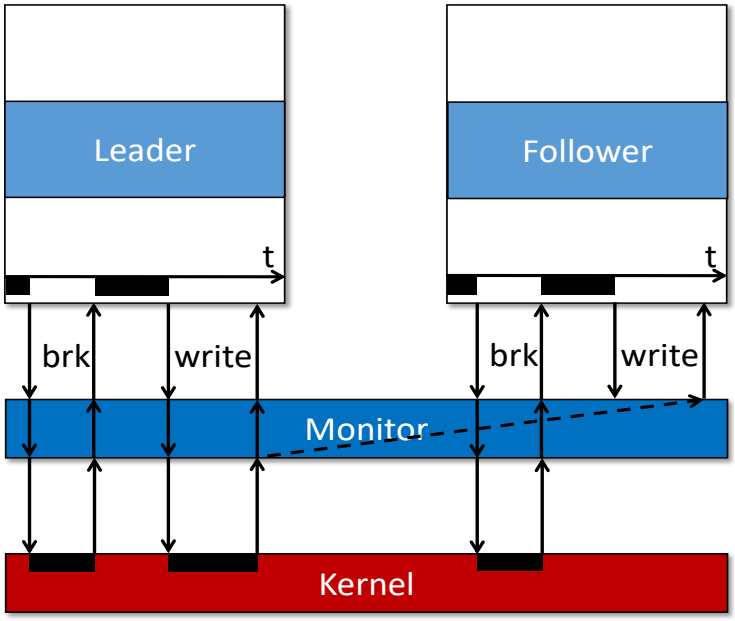


Figure 2.1: I/O Replication for Transparency.

[1]Communication via shared memory is not visible at the system call interface, so most NVX systems prevent mapping of shared memory regions.

2.1.3 Kernel Space versus User Space

The most important choice for an NVX monitor design is its placement in the software stack. The monitor interacts with the application variants frequently (*e.g.*, system call level). Consequently, it is of vital importance for this interaction to incur minimal overhead. However, we show that reducing the overhead without sacrificing security is challenging.

Existing NVX systems monitor program variants, stop them at rendezvous points (*e.g.*, system calls), look for observable divergences, and continue in case of equivalence. Consequently, we want to minimize the time spent for monitor-variant interaction. One obvious design, which the authors of one of the first NVX systems [18] used, is to implement the monitor in kernel space (shown as K-IP in Figure 2.2). This design provides good performance, however it requires significant modifications to the kernel/or kernel modules, which are both difficult to maintain and increases the Trusted Computing Base (TCB).

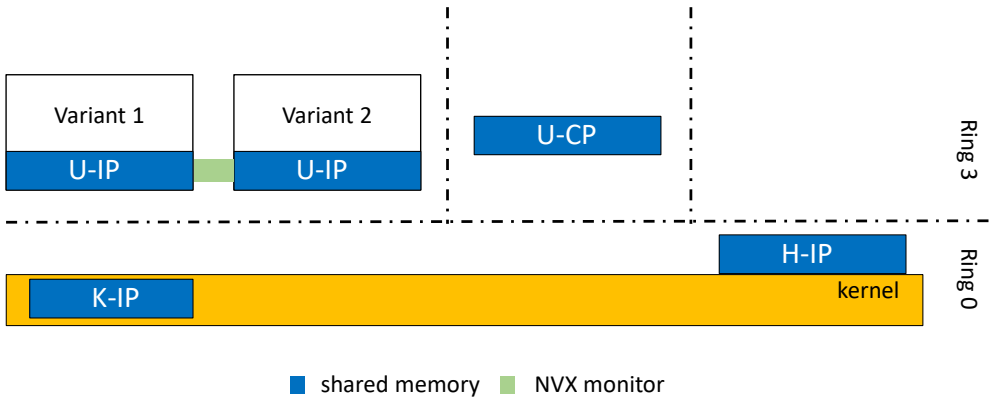


Figure 2.2: Different NVX monitor designs.

Another interesting choice (from a security perspective) is putting the monitor in user space, as a seperate process (shown as U-CP in Figure 2.2). Several previous solutions used the `ptrace` API, a debugging infrastructure provided by the Linux kernel, to build their user space monitors [79, 12, 47, 33, 61, 62]. `ptrace` gives the ability to the `tracer` process to stop the `tracee` at the start and end of every system call. Also since the `tracee` is at a different process, it is protected by the process isolation boundary provided by the operating

system. Therefore, a corruption in the program variant cannot corrupt the monitor itself. However, this comes at the cost of expensive context switching [78]. NVX systems that place the monitor in a separate process (cross-process monitor), provide maximum security but are not suitable for applications that use system calls frequently, such as web servers. For DMON, initially we chose this design since we expected network overhead (needed for inter-monitor communication) to be the performance bottleneck. We show in Chapter 7.1 that this assumption was wrong, if suitable network protocols and modern network adapters are used.

As a notable exception, we want to briefly talk about MvArmor [40]. This system is implemented at the hypervisor level (shown as H-IP in Figure 2.2), using a modified version of Dune [7]. From a security perspective, this is better compared to implementing the monitor in kernel space. However, it uses specific hardware (Intel VT-x extensions), which may not be available on all platforms. In addition, hypervisors are complex, difficult to maintain and significantly increase the Trusted Computing Base (TCB).

Table 2.1: Classification of NVX systems.

NVX Design	User Space	Kernel Space	Hypervisor
Cross-Process	DieHard [8], GHUMVEE [79] Cavallaro [12], Orchestra [61] Tachyon [47], Mx [33], **DMON** [80]	–	MvArmor [40]
In-Process	Varan [34]	N-Variant Systems [18]	–
	Bunshin [82]	kVMX [84]	–
Hybrid	**ReMon** [78], **DMON++**	–	–

2.1.4 Cross-Process versus In-Process

The high overhead of context switching motivated the authors of Varan [34] to place the monitor inside the variant's address space (this is shown as U-IP in Figure 2.2). Varan is a reliability-focused, in-process monitor. A shared memory segment, called a ring buffer is used to pass system call metadata and results required for monitoring and replication purposes.

Consequently, Varan's overhead is minimal even for applications that perform systems calls frequently, *e.g.*, web servers.

However, from a security perspective, a U-IP design is vulnerable to persistent attackers. First of all, an attacker can corrupt the monitor by corrupting the program variant, since they are in the same address space and there is no isolation between the program variant and the monitor. Secondly, Varan uses selective binary rewriting to rewrite all *syscall* instructions with jumps to custom system call handlers. However, this does not ensure that all of the system calls are monitored, since an attacker could access unaligned *syscall* instructions (unaligned *syscall* instructions are not rewritten by binary rewriting). In addition, an attacker can also tamper with the ring buffer, which is used for passing system call arguments and results, interfering with monitoring and replication logic.

2.2 ReMon – A Hybrid NVX system

Most security-oriented NVX systems such as cross-process monitors, sacrifice performance for security. On the other hand, reliability-oriented NVX systems such as in-process monitors, trade security guarantees for higher performance. In 2016, we developed a *new*, hybrid NVX system called ReMon [78] that combines the advantages of both in-process (U-IP) and cross-process (U-CP) monitor designs. More specifically, all *sensitive* system calls that could potentially corrupt the monitor are handled by the cross-process monitor (CP-MON), while *non-sensitive* system calls are redirected to a fast in-process monitor (IP-MON) (see Figure 2.3). ReMon uses the existing CP-MON GHUMVEE [79] to handle *sensitive* system calls.

ReMon's design is motivated by the fact that a security policy of monitoring all system calls is overly conservative [23, 60]. Many system calls cannot affect any state outside of the

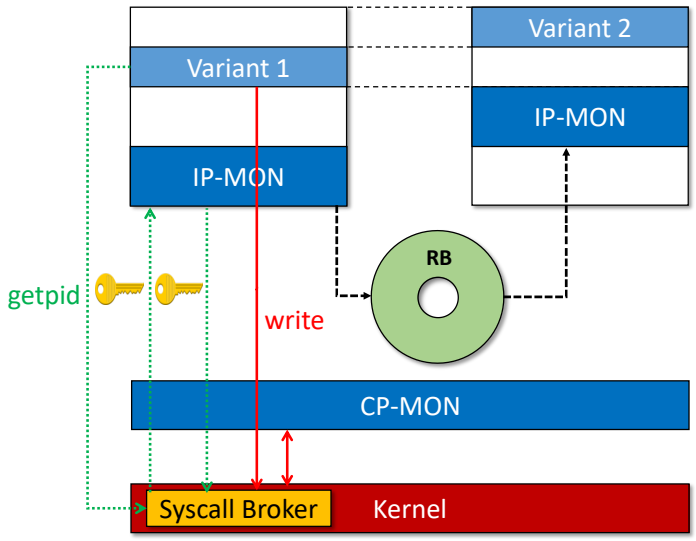


Figure 2.3: ReMon's High-Level Design.

process making the system call. Only a small set of *sensitive* system calls are potentially useful to an attacker. Thanks to the IP-MON component, ReMon supports configurable relaxation policies that allow *non-sensitive* calls to execute without being monitored. Similar to Varan [34], IP-MON uses a shared memory segment, called replication buffer (RB), to replicate system call results to follower variants. The relaxation policies are inspired by the classification of system calls used in OpenBSD [54]. Our policies are demonstrated in detail in Table 2.2.

2.2.1 Securing the Design

When the application wants to execute a system call, it goes through the kernel, and our minimal syscall broker decides if it will be forwarded to CP-MON or IP-MON. The syscall broker creates a valid one-time authorization token for every system call that is about to forward to IP-MON. As only the syscall broker can generate valid tokens, this mechanism

Level and description	*Unconditionally allowed calls*	*Conditionally allowed calls depending on*	
		file type	*op type*
BASE_LEVEL Read-only calls that do not operate on file descriptors and do not affect the file system.	gettimeofday, clock_gettime, time, getpid, gettid, getpgrp, getppid, getgid, getegid, getuid, geteuid, getcwd, getpriority, getrusage, times, capget, getitimer, sysinfo, uname, sched_yield, nanosleep		
NONSOCKET_RO_LEVEL Read-only calls on regular files, pipes, and non-socket file descriptors; read-only calls from file system; write calls on process-local variables.	access, faccessat, lseek, stat, lstat, fstat, fstatat, getdents, readlink, readlinkat, getxattr, lgetxattr, fgetxattr, alarm, setitimer, timerfd_gettime, madvise, fadvise64	read, readv, pread64, preadv, select, poll	futex, ioctl, fcntl
NONSOCKET_RW_LEVEL Write calls on regular files, pipes, and other non-socket file descriptors.	sync, syncfd, fsync, fdatasync, timerfd_settime	write, writev, pwrite64, pwritev	
SOCKET_RO_LEVEL Read calls on sockets.	read, readv, pread64, preadv, select, poll, epoll_wait, recfrom, recvmsg, recvmmsg, getsockname, getpeername, getsockopt		
SOCKET_RW_LEVEL Write calls on sockets.	write, writev, pwrite64, pwritev, sendto, sendmsg, sendmmsg, sendfile, epoll_ctl, setsockopt, shutdown		

Table 2.2: Monitoring Relaxation Policies.

forces every unmonitored system call to go through syscall broker. At the same time, it also ensures that IP-MON can only execute unmonitored system calls if it is invoked by the syscall broker and it is invoked through its intended entry point. This mechanism is, in essence, a form of Control-Flow Integrity [5]. It also allows us to hide the location of the RB, thereby preventing the RB from being accessed from outside IP-MON. Protecting the RB is of critical importance to the security of our NVX system, as it is used for replication of system call results.

The syscall broker loads the RB pointer and the token into designated processor registers whenever it forwards a call to IP-MON, and IP-MON is designed and implemented in such

a way that it does not leak these sensitive values into user space-accessible memory. First, we compile IP-MON using `gcc` and use the `-ffixed-reg` option to remove the registers we reserve for the RB pointer and authorization tokens from gcc's register allocator. This ensures that the sensitive values never leak to the stack, nor to any other register. Second, we carefully crafted specialized accessor functions to access the RB. These functions may temporarily load the RB pointer into other registers, *e.g.*, to calculate a pointer to a specific element in the RB, but they restore these registers to their former values upon returning. We also force IP-MON to destroy the RB pointer and authorization token registers themselves upon returning to the system call site.

Hardening IP-MON However, IP-MON is not resilient to bugs. An attacker could potentially find and use bugs in IP-MON's code to divert control flow to a malicious function that could leak the RB pointer or authorization token. Generally, an attacker would accomplish this by overwriting the target of an indirect jump instruction. We use inlining of `glibc` [25] functions to avoid indirect control flow instructions from IP-MONs system call entry point.

However, inlining `glibc` functions proved challenging. The optimized versions of these routines are implemented in hand-written assembly. As a first step, we needed to make sure that we correctly pass function parameters from the C/C++ code to the inline assembly and that we correctly copy function results from the inline assembly to the C/C++ code. In addition, in hand-written assembly it is common to use special symbols called labels. Labels are assigned the current value of the active location counter and serve as instruction operands. Unfortunately, the developers of these routines used global labels, which should have unique names. Several of these functions share names in their labels; consequently, we were not able to inline all of the functions together. Our solution to the aforementioned problem was to change all global labels to local labels which are not restricted to have unique names (see Appendix A for implementation details).

2.2.2 Server Benchmarks

We evaluated IP-MON on popular server applications including the `Apache` web server, `Thttpd`, `Lighttpd`, as well as `Beanstalkd`, `Memcached`, `Nginx` and `Redis`. Server applications are frequent attack targets and often run on *high-end* multi-core machines with idle CPUs that could be used by NVX systems to run variants in tandem. Another reason that we decided to evaluate IP-MON using server applications is the fact that servers typically represent workloads with a high frequency of system calls; making them ideal to show how efficient IP-MON is compared to a full cross-process monitor *e.g.*, GHUMVEE [79]

We tested IP-MON by running a benchmark client on a separate machine that was connected to our server via a local gigabit link with latency less than 0.5 ms. We used the exact same client and server configurations as previous work. Figure 2.4 shows our results. For each benchmark, we measured the overhead that IP-MON introduces when running between two and seven variants side by side with the spatial exemption policy at the `SOCKET_RW_LEVEL` (refer to Table 2.2 for details). We also show the overhead for running two variants with IP-MON disabled.

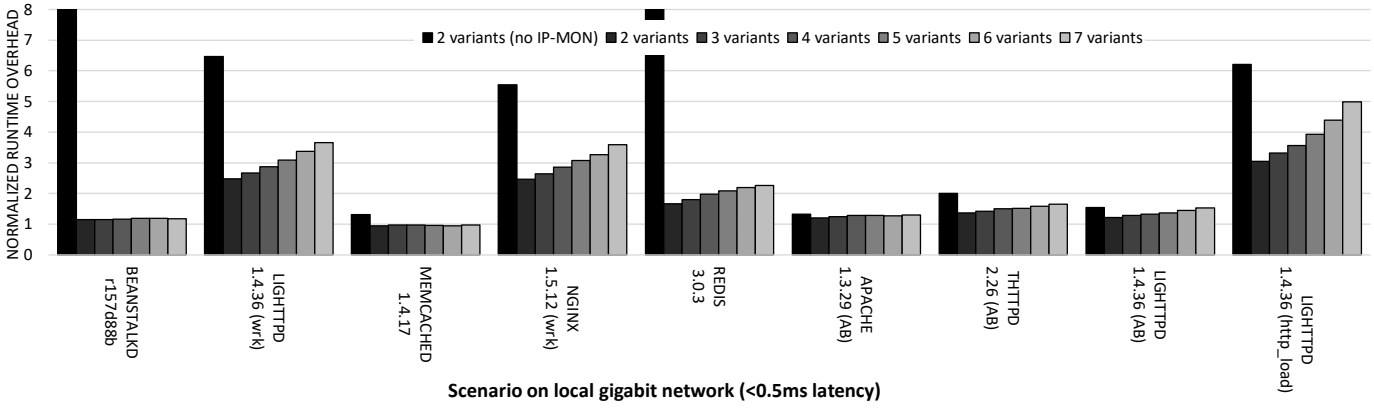


Figure 2.4: Server benchmarks for 2-7 variants with IP-MON and 2 variants without IP-MON.

Chapter 3

DMON – A Novel N-Variant System

Existing NVX systems have been particularly effective at stopping attacks that rely on knowledge of the target's absolute virtual address space layout (*i.e.*, code-reuse exploits whose payloads include absolute pointer values) [18, 10, 78], as well as attacks that attempt to acquire that knowledge through information leakage [46]. However, these systems are not resilient to Position-Independent Return-Oriented Programming (PIROP) attacks [26] and certain Data-Oriented Programming (DOP) attacks [36], which build on knowledge of the program's internal geometry (*e.g.*, relative data/instruction layouts) and/or data representation. The main reason is that in previous NVX systems all the variants run on the same machine as shown in Figure 3.1(a). Thus, the amount of diversity that such systems can achieve is limited to what a single platform can offer.

On the other hand, binaries targeted at multiple different platforms have an inherent diversity that comes naturally from differences in calling conventions, instruction set architectures, endianness, system call interfaces and available hardware features. This motivated us to design DMON, the first system that combines ISA and ABI heterogeneity with N-Variant Execution. DMON distributes the execution of a set of variants over a heterogeneous set of

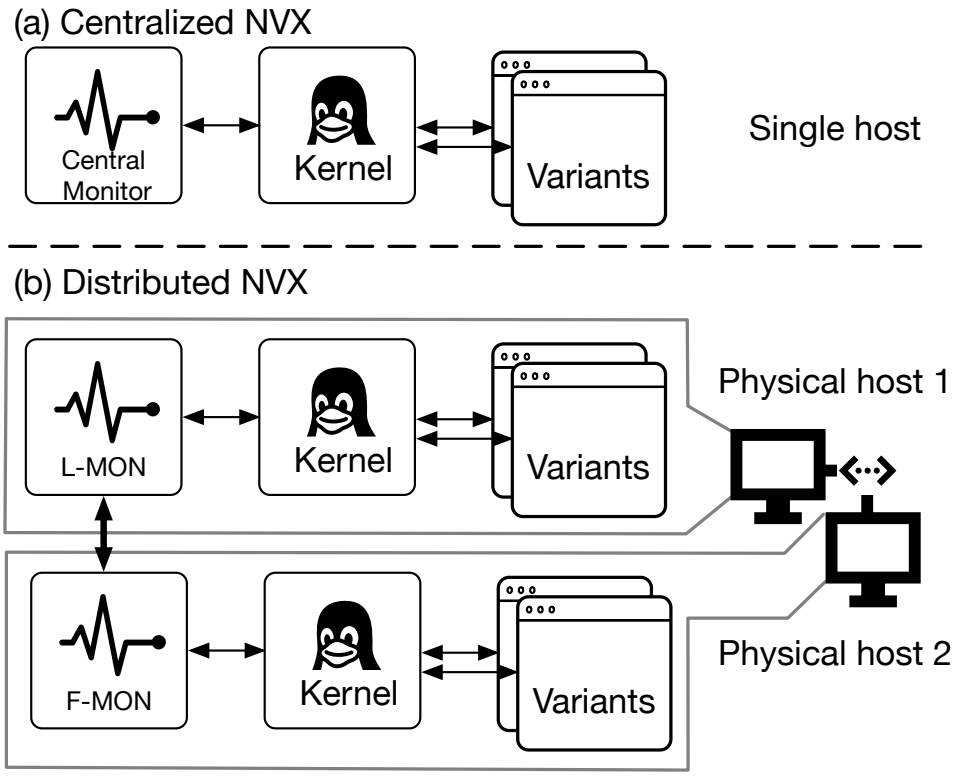


Figure 3.1: Centralized versus Distributed NVX designs (two variants).

physical machines as shown in Figure 3.1(b). DMON provides a natural resilience against memory exploits as it forces adversaries to develop exploits that work simultaneously against multiple ISAs, ABIs, system call interfaces, and hardware features.

3.1 DMON Design

DMON orchestrates and supervises the execution of a set of diversified program variants running natively on machines that differ in their instruction set architecture. Like many other NVX systems, DMON uses a leader/follower model for I/O replication. The designated leader variant is the only variant allowed to perform externally observable I/O operations such as sending or receiving data from a network socket. DMON forces follower variants to skip these I/O operations and instead provides them with the leader's I/O results as a way

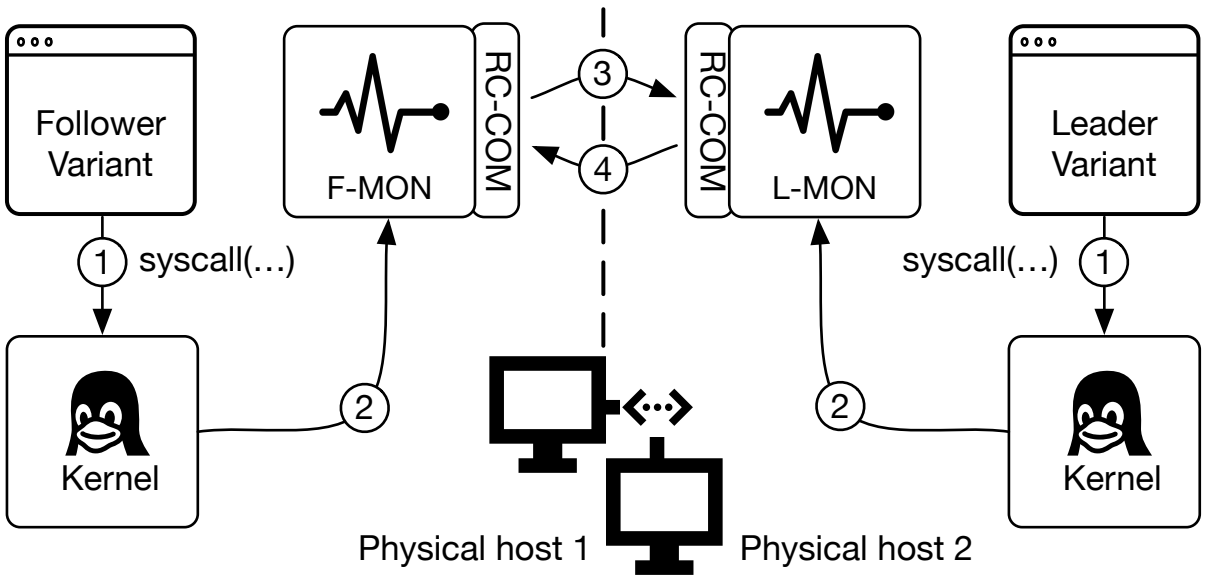


Figure 3.2: DMON's basic components and interactions.

of virtualizing I/O.

Similar to other security-focused NVX systems such as ReMon [78] and MvArmor [40], DMON classifies system calls based on their security sensitivity. Security *sensitive* system calls, that may be useful for attackers, are executed in lockstep. Whenever the variants attempt to execute a *sensitive* system call, DMON ensures that the variants can neither enter the system call routine, nor exit from it until DMON has ensured that all variants have reached equivalent states. We distinguish between the following components of a running DMON system:

1. **Leader Variant** Only the designated leader variant is allowed to perform externally observable I/O. As in any other NVX system, DMON requires that there is exactly one leader variant. The leader designation is fixed. Leader variants cannot become followers and vice versa.

2. **Follower Variants** Follower variants skip externally observable I/O operations and use the leader's I/O results instead. DMON supports any number of such variants.

3. **Monitors** The monitors are responsible for starting the variants, supervising their execution, exchanging system call metadata (system call numbers, arguments, and results), performing security checks, and enforcing lockstep execution. DMON uses two types of monitors: the (single) L-MON monitor supervises the leader variant, while every follower variant is supervised by its own F-MON monitor.

4. **RC-COM** A reliable communication component used to exchange system call metadata between the monitors. By separating the communication logic into its own abstraction layer, we have enabled the monitors to communicate over a variety of communication channels (*e.g.*, loopback interfaces vs network cards, different network protocols, etc.).

These components interact whenever the variants execute system calls, as shown in Figure 3.2. Whenever a leader or follower variant attempts to enter or exit from a system call (①), the corresponding L-MON or F-MON interrupts and suspends the variant, reads the call number of the interrupted system call, and invokes a specialized handler routine within the monitor process (②), which implements the monitoring (also called cross-checking) and replication logic for that system call.

The monitors use cross-checking handlers when they interrupt variants upon entering a system call. In F-MON, the cross-checking handler gathers information about the variant's state, sends this information to L-MON (③), and waits for L-MON to confirm that the follower variant is in a state equivalent to the leader variant (④). In L-MON, the cross-checking handler waits for incoming state information from F-MON, compares that state information with the leader variant's state, and informs F-MON about the results of the comparison.

The state information consists of system call numbers and arguments, with the latter often consisting of pointers to complex data structures (*e.g.*, I/O vectors). The cross-checking

handlers serialize these corresponding data structures and append the serialized data to the state information, thereby allowing L-MON to check the variant states for deep equivalence (two data structures are deeply equivalent when the raw data they contain is identical, even though the data or the data structures may be stored at different addresses). If the variant states do not match, DMON takes that as a sign of potential compromise and aborts execution to protect the host system.

Naive cross-checking of these variant states triggers false alarms for divergent behavior because the system call interfaces, calling conventions, etc. differ across platforms. DMON transforms system call states to platform-independent states before comparing them, to avoid alarms for the expected platform differences (see Chapter 4.2.3).

If the states match, the cross-checking handler allows the leader variant to proceed and to enter the kernel space system call routine. The follower variants can also proceed, but may (optionally) see their system call number replaced by that of the `sys_getpid` routine in case they attempt to perform an externally observable I/O operation. This mechanism for skipping system calls was also used in prior work [61].

The monitors use replication handlers when they interrupt variants that return from a system call. Replication handlers for I/O system calls broadcast the system call results from the leader variant to the followers. Replication handlers for other system calls are generally no-ops.

3.2 DMON Infrastructure

Prior work often used a central monitor process which simultaneously supervised all of the variants [61, 79, 10]. Subsequent research showed that this centralized model was overly focused on simplicity and security at the expense of performance, and suggested various

designs in which each variant was supervised by a dedicated monitor instance [34, 78, 40, 82, 46, 84]. This dedicated monitor instance could be loaded directly into the variants' address spaces, thereby sacrificing the isolation between the variants and the monitor for greatly reduced variant-monitor communication overhead.

DMON combines elements of both designs. Since we run ISA-heterogeneous variants on different physical machines, we cannot use a single (central) monitor that attaches locally to all variants. Instead, we use a dedicated monitor for each variant and run the monitor on the same machine as the variant it supervises. Our design does, however, enforce strict isolation between the variant and its monitor by running the monitor as a separate process that attaches to the variant using the `ptrace` API [1].

Each monitor includes an RC-COM, which exposes our inter-monitor communication API. By separating the low-level communication logic from the monitor, we were able to implement and compare various communication mechanisms (see Chapter 5.1).

3.2.1 Monitor at system call interface

NVX systems that are implemented in user space need a way to intercept system calls. In-process monitors use binary rewriting and/or kernel patches to replace system call entry points with jumps to custom handlers. The vast majority of cross-process monitors use the `ptrace` API to intercept system calls.

Security-oriented NVX systems typically run program variants in lockstep, making sure that they all reach specific rendezvous points (*e.g.*, at system call level). `ptrace` is a debugging infrastructure provided in Linux platforms to monitor user space programs at system call level. NVX systems leverage `ptrace` API to stop applications before and after executing a system call.

The monitor ensures that all variants have equivalent system call states before executing a system call. For non-pointer arguments, we just compare bit-by-bit their values, and for pointer arguments we compare the data that they refer to. In Chapter 4.2.3, we show that this is not enough for a heterogeneous environment and that we need to redefine the equivalence between system call states.

In case the system call arguments of two variants are not equivalent before entering a system call, DMON takes action and stops their execution. However, there are reliability-oriented NVX systems [59] that are able to *tolerate* some expected divergences at system call level using rules written in a Domain-Specific Language (DSL). The salient idea is that we can automatically extract these rules from program traces or manually write them ourselves.

Stopping at the return of a system call also ensures transparency of our NVX system (see Chapter 2.1.2). DMON copies the system call results of the leader variant to all the followers, when needed, using the network and monitor-variant data transfers as described in Chapter 3.2.2.

3.2.2 Monitor-Variant Data Transfers

In-process monitor designs can directly read data from or write data to the variant's address space, needed for monitoring and replication. In a cross-process design like DMON though, we need to use some Inter-Process Communication (IPC) mechanism for data transfers from/to the variant's address space.

One method to read from the memory of the processes is to call `ptrace` with `PTRACE_PEEKDATA` when the variants are suspended. `PTRACE_POKEDATA` can similarly be used to write to the variants. Because `ptrace` only returns four bytes at a time for 32-bit systems and eight bytes for 64-bit systems, `ptrace` has to be called several times to read/write a large block

of memory from/to the variants address spaces. Every call to `ptrace` requires a context switch from the monitor to the OS kernel and back, which makes this technique inefficient for reading large buffers.

In DMON, we use the `process_vm` API [2] for monitor-variant data transfers as it was used in previous NVX systems [79]. This data transfer technique is far faster than `ptrace`-based data transferring. However, it is not suitable for reading string arguments, since we do not know their size beforehand. In this particular case, we use PTRACE_PEEKDATA and read the string word by word until we find the `end_of_string` character.

Chapter 4

Inconsistencies and False Alarms

N-Variant Execution is based on the underlying assumption that all variants are getting the same input. For example, the result of a system call *e.g.*, `sys_read` could be the input for another system call *e.g.*, `sys_write`. Our leader/follower replication mechanism, described in Chapter 2.1.2, copies system call results from the leader variant to the follower variants, ensuring that in this case, all variants are getting identical input.

However, NVX systems need to deal with sources of input that do not directly use system calls, *e.g.*, virtual system calls, and as well as sources of non-determinism, *e.g.*, inter-thread communication in multithreaded applications. These could lead to divergences at the system call interface, which will be detected by our monitor. In this chapter, we briefly discuss existing solutions to the aforementioned problems and DMON's limitations.

Finally, we show that our distributed, heterogeneous setting causes *new* false positive detections (or false alarms). We categorize these *benign* divergences and we propose a *novel* system call metadata transformation, called Platform-Independent State Canonicalization (PISC), to tolerate and/or eliminate them without sacrificing the security guarantees of DMON.

4.1 Inconsistencies and False Alarms in Centralized NVX systems

Existing, centralized NVX systems have lifted completely different design choices that greatly affect their security and performance guarantees. However, they face similar inconsistencies and false positive detections.

4.1.1 Virtual System Calls

On most architectures, Linux loads a **Virtual Dynamic Shared Object (VDSO)** [74] or **vsyscall** page into the address spaces of all user space programs. These executable code pages expose a number of so-called virtual system calls, which allow the program to execute certain system calls (*e.g.*, `sys_gettimeofday`) without mode switching into kernel-space. Consequently, system call interception mechanisms used by NVX systems (either cross-process or in-process) would not be able to catch these virtual system calls, leading to inconsistencies, *e.g.*, for random number generators.

Most NVX systems hide, replace, or disable the VDSO and vsyscall page because virtual system calls are invisible to the monitor, and therefore bypass the replication mechanism. To eliminate the inconsistencies caused by VDSO, we patched the C library our variants link against so that the library never uses virtual system calls. Operations such as `gettimeofday` are therefore always visible to our monitors. Our patches are minimal and we were able to patch and use two popular C library implementations; `glibc` [25] and `musl libc` [50] (see Appendix B).

4.1.2 Asynchronous Signals

Asynchronous signals are beyond the scope of this dissertation. This is not a fundamental limitation of our approach but a matter of additional engineering effort. Signal handling mechanisms would be implemented as has already been described in earlier work [61, 34, 79]. Existing solutions, force synchronous signal delivery of asynchronous signals to all variants, and/or restarting system calls when needed.

4.1.3 Multithreading

Running multithreaded variants currently may trigger divergences and false positive detections. Earlier work describes techniques to support variants in which the threads do not communicate directly [34, 40, 82, 46]. These techniques can directly be incorporated into our distributed heterogeneous approach. Supporting variants in which the threads *do* communicate directly is more difficult. The current state-of-the-art is to capture the order in which the leader variant executes thread synchronization instructions, and to replay this order in the follower variants [77]. Incorporating this mechanism into our distributed heterogeneous NVX design would require substantial engineering effort.

4.1.4 Address-Dependent Behavior

Similarly, there might be divergences in programs featuring address-dependent behavior. If a program inserts objects into a binary tree based on their memory addresses, and subsequently prints out that tree, the output would likely differ across variants. Earlier work describes several similar cases [79]. We are not aware of any automated solutions for identifying and neutralizing such address-dependent computations, so this is currently an open problem that applies to all NVX systems.

4.1.5 Shared Memory

Modern Linux-based operating systems support two APIs for sharing memory between processes, the System V IPC API and the file mapping API using `sys_mmap` and *family* system calls. Shared memory is used as a fast Inter-Process Communication (IPC) mechanism. Unfortunately, reading from or writing to the shared memory segment happens at user space without invoking system calls. Consequently, NVX systems can not intercept accesses to shared memory and guarantee that all variants read the same data that are written by an external entity.

A potential solution is to revoke access rights to shared memory and emulate shared memory accesses using custom signal handlers. However, this is proven to be slow [38]. We could augment this solution with a sound (but not complete) static analysis to automatically insert cross-checking code at shared memory accesses entry-points and also perform replication when needed. Another option is to completely revoke accesses to shared memory and hope that the application has a back up IPC mechanism such as pipes or sockets. This assumption was true in the past for most of the applications. However, currently the majority of programs are solely depended on shared memory for IPC, due to its performance benefits. Shared memory support in NVX systems is an open problem and is outside the scope of this dissertation.

4.2 False Alarms in a Heterogeneous NVX System

DMON runs variants in different physical machines with different architectures, in order to increase the level of internal diversity between the simultaneously running variants that can be supported. Unfortunately, ISA/ABI-heterogeneity affects the system call interface, resulting in false alarms (or divergences) in an NVX system. We studied the ABI specification

documents and analyzed actual system call traces of both trivial and complex applications running on ARMv7, ARMv8, i386, and x86-64 CPUs to understand and anticipate false divergences arising from ISA/ABI-heterogeneity. Our findings showed that it is not trivial to deal with these divergences, since architecture greatly affects the system call interface.

4.2.1 ISA-Heterogeneity

An underlying assumption of NVX is that the program variants will behave identically if i) they are built from the same source code, and ii) they receive equivalent, benign inputs. This assumption no longer holds in our setting, where we run variants on processors with different Instruction Set Architectures (ISAs). Differences in the endianness, register and pointer width — and even the available system calls could lead to observable (yet benign) differences in the variants' behavior, which would all cause false alarms in a traditional NVX system. As a result, we designed DMON to tolerate any expected divergences arising from the heterogeneous-ISA setting.

4.2.2 ABI-Heterogeneity

Aside from obvious differences such as different instruction opcodes and encoding, endianness, and register/pointer width, ISA-heterogeneous variants often also differ in more subtle ways because the OS maintainers impose a set of conventions for all binary programs compiled for the target ISA. The Application Binary Interface (ABI) documents rules such as sizes of primitive data types, how structs are packed, padded, and aligned, how function callers can pass arguments to their callees, etc. Many of these conventions also affect the program behavior as observed from the system call interface, and we therefore had to carefully design DMON to take ABI differences into account when comparing variant behavior.

4.2.3 Platform-Independent State Canonicalization

In traditional NVX systems, all program variants are compiled for the same target architecture and execute on a single machine. In DMON, on the other hand, individual variants run on different physical machines and thus the variants may target different ISAs and ABIs. The target ISA and ABI both affect a program's behavior as observed from the system call interface. ISA/ABI-heterogeneity therefore challenges the core assumption that variants will behave identically when provided with identical inputs, as long as the inputs do not trigger a program bug nor is the program being attacked.

We studied the ABI specification documents and analyzed actual system call traces of both trivial and complex applications running on ARMv7, ARMv8, i386, and x86-64 CPUs to understand and anticipate benign behavioral divergences arising from ISA/ABI-heterogeneity. We incorporated our findings into a transformation technique, called Platform-Independent State Canonicalization (PISC), that eliminates benign divergences that stem from ABI/ISA heterogeneity, by converting system call states to platform-independent states before cross-checking. We summarize the different classes of expected divergences and their causes in Table 4.1.

Note that PISC is also applicable to traditional non-distributed NVX systems. For example, we could combine PISC with existing NVX systems to support i386 and x86-64 variants that run on the same x86-64 machine. Furthermore, PISC could be deployed along existing NVX systems to support heterogeneous variants in future heterogeneous-ISA chip multiprocessors.

System Call Numbers. The system call numbers often differ between ABIs. The `sys_read` system call, for example, has system call number 0 on x86-64 platforms and on ARMv7 platforms that implement the old ARMv7 ABI (i.e., `arm-linux-gnu`), 3 on i386 platforms, and $0x900003$ on ARMv7 platforms that implement the new ARMv7 ABI (i.e.,

Divergences source	*Affected system calls*	*PISC Action*
SYSTEM CALL NUMBER System call numbers differ between ABIs and/or identical code is translated to slightly different system calls.	Every system call	PISC keeps its own mapping that is aware of the ABIs/ISAs diversity.
SYSTEM CALL ARGUMENTS Same code is translated to different system calls with similar functionality and diverse arguments.	stat, fstat, fstatat, newfstatat, open, openat, rmdir, unlink, unlinkat, chown, fchownat, mkdir, mkdirat, pipe, pipe2, dup2, dup3, fork, clone, epoll_wait, epoll_pwait	Transformation of system call metadata into platform-independent state.
STRUCT LAYOUT Struct layout diversity caused from different ABIs.	System calls that use at least one argument that points to a struct.	PISC keeps its own "shadow" structs and deals with conversions.
FLAGS AND MODES File flags and access modes used as system call arguments may be mapped to different integer values.	System calls that use integer arguments as file flags or access modes.	PISC keeps its own mapping that is aware of the ABIs/ISAs diversity.

Table 4.1: Categories of expected divergences.

`arm-linux-gnueabi`). Directly comparing these numbers will cause a benign divergence. To resolve this issue, we define a platform-independent identifier for each system call and create a mapping between system call numbers and the identifiers. DMON uses these platform-independent identifiers to cross-check system call states.

System Call Arguments and Struct Layout. System call arguments are not necessarily bit-for-bit identical across ABIs, even when the arguments are in fact the same. This is

particularly true for C structs which may be packed differently (i.e., there may be different numbers of padding bytes between the struct fields) or have different sizes depending on the ISA and ABI. To allow for bit-by-bit comparisons of such structs, PISC converts them to an internal "shadow" type that is carefully specified and/or annotated so it has the same layout on all platforms.

Use of Different System Calls. Beyond minor differences in system call numbers and arguments, heterogeneous-ISA variants may use different system calls altogether, because not all system calls are available on all platforms. ARMv8 kernels, for example, do not implement the `sys_open` system call. ARMv8 variants therefore always use `sys_openat` to open a file. `sys_openat` is similar to `sys_open`, but does have an additional argument that can hold the file descriptor of a directory. If the `pathname` argument of the `sys_openat` is relative, then it is interpreted relative to the directory specified in the additional argument.

x86-64 kernels, on the other hand, implement both `sys_open` and `sys_openat` and x86-64 variants regularly use both APIs. Consequently, we may see divergences in a setup where an ARMv8 and x86-64 variant try to open the same files. PISC deals with these divergence by transforming the system call states for similar system calls into generic platform-independent states, prior to cross-checking. In this concrete example, PISC would fully resolve the paths the variants are trying to access and then build system call states corresponding with the invocation of a `sys_openat` call, regardless of whether the variant called `sys_open` or `sys_openat`.

System Call Flags. There are several system calls that accept flags as their arguments. These flags can specify file access modes, file statuses, etc. The integer values assigned to these flags may differ across ABIs. Consequently, cross-checking these flag values may trigger false positive detections. Once again, PISC deals with these divergences by mapping all flag

values to an internal platform-independent value prior to cross-checking.

Chapter 5

Removing the performance penalty

Running DMON comes at the cost of *expensive* network communication, needed for cross-checking and replication logic. We provide multiple implementations of the communication channel with different performance guarantees. In addition, we present optimizations that reduce the amount of cross-checking and replication, eliminating inter-monitor communication when possible.

5.1 Inter-Monitor Communication

F-MON and L-MON communicate whenever the variants execute a system call. This exchange may include system call numbers, serialized system call arguments, system call results, or instructions on how to proceed from a system call entry point (see Chapter 3.1). In many cases, particularly when the system call being executed is deemed security-sensitive, communication must happen synchronously. For instance, L-MON cannot allow the leader variant to proceed past a system call entry point until all instances of F-MON have serialized the state of their corresponding variant, and until they have sent this state to L-MON.

F-MON needs to wait even longer as it cannot allow the follower variants to proceed until L-MON has compared the variant states and it has received L-MON's confirmation that the states match.

For good performance, DMON therefore requires a reliable inter-monitor communication channel with minimal latency and high bandwidth. There are many ways to realize such a channel. We experimented with various designs of this communication channel and implemented them in our RC-COM, which exposes the inter-monitor communication API to our monitors.

Network Protocol Choice. The most obvious protocol that meets our reliability demands is TCP, which we used as the basis for our first implementation of RC-COM. However, even with extensive tuning, our TCP-based implementation had poor throughput and high latency. As an alternative, we therefore used ENet, a lightweight UDP-based protocol that also offers reliable in-order and error-free data transfer [27]. Our performance evaluation confirms that ENet is more efficient than the TCP-based implementation that uses the standard TCP/IP stack.

User Space Networking. Besides the networking hardware, the operating system also affects the communication bandwidth and latency. When a network adapter receives a packet, for example, the OS first stores the packet in a kernel space buffer, before copying it into the receiving application's memory and transferring control to the application. These extra copy operations can be avoided with techniques such as Remote Direct Memory Access (RDMA). RDMA allows two communicating peers to read or write directly from or to the other peer's application memory, thus bypassing the kernel's networking stack. We implemented an RDMA-based version of our RC-COM using Mellanox ConnectX 100 gigabit ethernet interfaces [48] and the Mellanox Messaging Accelerator user space networking

library [44].

5.2 Further Optimizations

To improve DMON's performance even further, we implemented several optimizations that can reduce the number of the data packets exchanged by our monitors during cross-checking and replication.

5.2.1 Asynchronous Cross-Checking

Our basic approach described in Chapter 3.1 adds considerable overhead to every system call invocation as every cross-check happens synchronously and requires at least two network round-trips; one for F-MONs to send the system call states of their supervised variants to L-MON, and one for L-MON to instruct F-MONs on how to proceed (abort or continue execution of the variant).

We developed a technique which we call *asynchronous cross-checking* to reduce this overhead. Inspired by previous work [78, 40], the idea is to classify system calls into three categories — highly *sensitive*, moderately *sensitive*, and non-sensitive — based on the system call number and/or arguments. With asynchronous cross-checking, highly *sensitive* system calls still execute in lockstep, as before. When F-MON deems a system call moderately *sensitive*, however, it still sends the system call state information to L-MON, but then immediately resumes execution of the supervised variant without waiting for a reply from L-MON. L-MON eventually receives the state information and may detect a divergence. In that case, L-MON will instruct F-MONs to abort execution through a separate error channel that is used only for this specific purpose. *Non-sensitive* system calls can execute without any cross-checking at all.

Note that this policy differs from the selective cross-checking described in previous work as DMON detects *all* divergences for *sensitive* system calls including moderately *sensitive* ones, whereas ReMon and MvArmor only detect divergences on invocations of highly *sensitive* system calls [78, 40]. There may, however, be a delay in the detection of divergent invocations of moderately *sensitive* system calls.

5.2.2 Selective Replication

Similar to cross-checking, replication also uses network for inter-monitor communication. We show that replication could be avoided for specific I/O operations and we call this technique selective replication.

Permissive Filesystem Access. Traditional NVX systems enforce replication for *all* I/O operations, regardless of the type of I/O resource being accessed. The system allows one variant to effectively perform the operation and it then replicates the results to the other variants. Even though this replication mechanism seamlessly provides identical inputs to all variants, it is not always necessary. Specifically, there is no need to replicate read accesses to read-only files that were identical on all physical machines when DMON started, as long as the files have not been modified while DMON was running. We refer to such files as *static* files and designed the cross-checking handlers for read-only operations such as `sys_read`, `sys_readv`, `sys_pread`, and `sys_fstat` so that all variants may (optionally) read *static* files directly from their local file system, thus bypassing the I/O replication.

For this optional optimization, DMON requires that the application's root directory has the same path name on all machines as well as identical content including sub-directories with the exception of executables and shared libraries.

Immutable State Caching. Many system calls, including `sys_getpid` read immutable state such as the process ID or thread ID. To avoid unnecessary replication, DMON caches the results of the first invocation of these system calls. DMON immediately cancels any subsequent invocations of these calls and returns the cached result instead. Note that, contrary to our other optimizations, immutable state caching cannot be disabled.

5.3 DMON Implementation

We implemented DMON for GNU/Linux. DMON runs natively on the x86-64 and ARMv8 architectures and supports variants compiled for these architectures. DMON also has partial support for ARMv7 and i386. Currently, our prototype has 35k lines of C and C++ code and supports variants compiled with the stock versions of gcc and Clang. We do, however, require the variants to link against our patched C library (see Chapter 4.1.1) and to be compiled with the same toolchain version.

System Call Support and Classification. DMON currently supports 100+ system calls. Adding support for additional system calls generally requires a trivial amount of engineering effort (typically less than 10 lines of code), as DMON defines helper macros to replicate and cross-check most types of system call arguments. Our helper macros resemble those used in ReMon [78], but differ from them as they automatically apply PISC, thus making our macros fully portable.

Table 5.1 shows an overview of these calls, and how DMON cross-checks and replicates them. The type of cross-checking depends on the security-sensitivity of the call (see Chapter 5.2). DMON always cross-checks highly *sensitive* system calls in lockstep. Moderately *sensitive* calls are checked asynchronously. *Non-sensitive* calls are not checked at all.

The type of replication depends on the kind of results the system call returns. DMON enforces replication for all I/O operations that are not reads from *static* files (see Chapter 5.2), and for all system calls that return mutable program state. Read operations from *static* files execute without replication if the permissive filesystem access optimization is enabled. System calls that must be executed by all variants and system calls that read cached immutable program state are not subject to any replication.

Cross-Checking	Replication		
	Always	**Possibly**	**Never**
Lockstep	fork, clone, socket, bind, listen, connect, accept4, accept, socket, socketpair, eventfd2, epoll_create, epoll_create1, chown, fchownat, pipe, pipe2		execve, mprotect, mmap, munmap, brk, exit_group, setuid, setgid, setsid, chdir, umask, prctl, chown, fchownat, rt_sigprocmask, rt_sigaction, rt_sigsuspend
Possibly Asynchronous	write, writev, pwrite64, send, sendto, sendfile, epoll_ctl, open, openat setsockopt, shutdown, epoll_wait, epoll_pwait, clock_gettime, time, gettimeofday, sysinfo, recvfrom, getsockname, getpeername, getsockopt	lseek, stat, fstat, fstatat, newfstatat, dup, dup2, dup3, close, fcntl, ioctl, read, readv, pread64, umask, uname	getcwd, mkdir, mkdirat, rmdir, unlink, unlinkat, sync, fsync, access, faccessat
None			getpid, getppid, futex, sched_yield, nanosleep, gettid, getgid, getegid, getuid, geteuid, arch_prctl, prlimit64, getpriority, madvise, sched_getaffinity, set_tid_address, getpgrp, set_robust_list

Table 5.1: Supported system calls and their classification.

Chapter 6

Security Analysis

6.1 Threat Model

Throughout the rest of this dissertation, we will make the following assumptions about the host system and the attacker. Our assumptions are consistent with related work in this area [78].

Host defenses We assume that the standard set of mitigations is in place on any of the physical machines DMON and the variants run on. Specifically, we assume that Data Execution Prevention (DEP) is used, and that memory pages are therefore never writable and executable at the same time. DEP therefore rules out direct code-injection attacks. Likewise, we assume that all of the host systems have Address Space Layout Randomization (ASLR) enabled. ASLR randomizes the base addresses of the main program executable and shared libraries, as well as the heap, stack, and any other mapped memory regions. Note that, even with ASLR in place, the NVX system can still override the base addresses of any region mapped into a variant's address space [76, 46].

Known and vulnerable target We assume that the protected application is known to the attacker, and that the attacker either has direct access to the variant binaries, or that the attacker can reproduce exact replicas of any of the target binaries for offline analysis. We further assume that the protected application has an arbitrary memory read/write vulnerability that the attacker knows how to trigger.

Remote attacker We assume that the attacker does not have direct physical access to any of the machines DMON (or the variants) run on. The attacker can only communicate with the protected application running on the leader machine via a remote communication channel such as a network socket. The followers are connected to the leader through a secure private network connection. The adversary can, therefore, not communicate with the follower variants. We also consider attacks on this private connection to be outside the scope of this dissertation. Because the attacker is remote, we also assume that any run-time secrets embedded into the variants (*e.g.*, randomized base addresses) are not known a priori. The goal of the attacker in this scenario is to take control of the leader variant, *e.g.*, by exploiting a memory-corruption vulnerability.

6.2 Evaluating DMON's security

Scope. NVX systems can prevent usage of *absolute* code addresses by adopting a technique called Address Space Partitioning (ASP) [18, 76, 46]. With ASP, the NVX system lays out the variants' address spaces in such a way that (i) the base addresses of executable code regions are randomly chosen, and (ii) no absolute memory address ever points to valid code in more than one variant. We refer the reader to earlier work for more details [18, 76, 46], and focus on evaluating the additional security DMON can provide through ISA/ABI-heterogeneity. Specifically, we show the extent to which ISA/ABI-heterogeneity prevents

concrete code-reuse and data-only attacks that cannot be easily stopped using existing NVX systems.

Analysis Targets and Configurations. We used four popular server applications — Nginx 1.14.2, Lighttpd 1.4.52, Redis 5.0.1, and ProFTPD 1.3.0 — as our analysis targets, which is in line with previous work on security-oriented NVX systems [78, 40, 82, 46]. We evaluated the security of a heterogeneous configuration with one program variant compiled for Intel x86-64 and one for ARMv7.

6.2.1 Code Layout Diversity

Existing NVX systems that deploy address space partitioning (ASP) can be bypassed using attacks that rely on partial overwrites of code pointers such as return addresses or function pointers [20, 26]. The basic idea is to force the program to produce a (number of) legal code pointer(s) at memory locations that the attacker can overwrite. The attacker then overwrites the least significant bits or adds arbitrary offsets to each of these code pointers, and thereby diverts the execution of the program to a series of attacker-chosen gadgets (*i.e.*, instruction sequences ending with indirect branches, such as return instructions). In the PIROP attack, for example, Göktas et al. exploited a vulnerability in the Asterisk communication server that allowed them to produce legal return addresses at an attacker-controlled position on the stack [26]. They then overwrote the least significant byte of each of these return addresses to build a PIROP gadget chain, which they then invoked by exploiting another vulnerability.

The reason why these attacks bypass existing NVX systems is because they do not require any information leakage (which the NVX system would detect), and because the same partial pointer overwrites can achieve the same results in each variant. In this section, we show that DMON makes these position-independent code-reuse attacks far more challenging because

ISA/ABI-heterogeneity substantially reduces the number of position-independent gadgets available to the attacker.

Position-Independent Gadget Availability. Position-independent gadgets are instruction sequences that can be *reliably* invoked by patching legal code pointers. We consider two ways to patch legal code pointers. First, an attacker could overwrite an offset variable that is later added to a code pointer in a pointer arithmetic operation. This primitive allows attackers to reliably invoke any gadget, as long as the internal layout of the target binary is known.

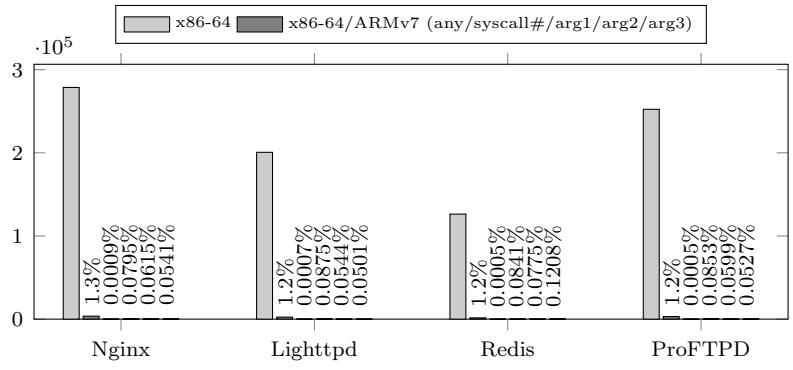
Second, the attacker could overwrite the least significant bits of a code pointer directly using a memory write vulnerability. This primitive is far less potent than the former, as it allows the attacker to overwrite only the 8 least significant bits (*i.e.*, one byte). Overwriting more than one byte is not possible unless the attacker knows the base address of the target binary because the ASP scheme randomizes all but the 12 least significant bits of each base address.

We compiled a list of the position-independent gadgets in both our x86-64 and ARMv7 binaries as follows. We first collected the addresses of (i) all instructions that immediately follow call instructions, and (ii) all address-taken functions in the program. The former is an approximation of the set of legal return addresses that could exist in the program's address space at any given point during its execution. The latter is the set of other code pointers that could be found in the program's memory. Combined, this list approximates the set of pointers that *could* potentially be patched by attackers to construct position-independent code-reuse payloads.

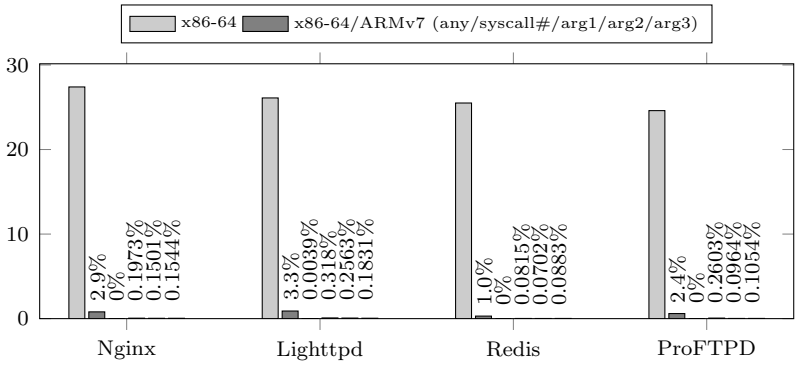
We then used Ropper [63] to generate lists of regular ROP gadgets consisting of 15 instructions or less. This, again, is consistent with related work [26]. Next, we combined the two lists for each binary as follows. For every code pointer in the first list, we calculated the (i)

addresses of all gadgets relative to the pointer, and (ii) absolute addresses of gadgets that only differ from the code pointer in their 8 least significant bits. The former is the set of gadgets reachable through offset overwrites, while the latter is the set of gadgets reachable through partial pointer overwrites.

Next we correlated the position-independent gadgets found for the x86-64 binary with those found for ARMv7. For each x86-64 gadget, we checked whether there is an ARMv7 gadget that can be reached using the same offset overwrite/partial pointer overwrite. We then eliminated gadgets whose absolute address or offset from the source code pointer is not 4-byte aligned, since code pointers patched in either way would be unaligned on ARMv7 and would trigger an unaligned instruction exception when the gadget is invoked.



(a) Offset overwrite



(b) Partial pointer overwrite

Figure 6.1: Number of position-independent code-reuse gadgets from each code pointer for each pointer patching strategy.

We collected 2553 code pointers from Nginx, 1988 code pointers from Lighttpd, 1732 code

pointers from Redis, and 4514 code pointers from ProFTPD. Figure 6.1 shows how many gadgets can be reached on average from each code pointer by offset overwrite and partial pointer overwrite attacks. In a traditional NVX system where all variants are compiled for Intel x86-64, all of the gadgets identified in the x86-64 binary would survive. In contrast, in all four of our target programs, and for both code pointer patching strategies, less than 3.3% of the gadgets survive in an NVX configuration with a x86-64 variant and an ARMv7 variant.

Table 6.1: Comparison of function and syscall conventions.

arch/ABI	**syscall#**	**arg1**	**arg2**	**arg3**	**arg4**	**arg5**	**arg6**	**arg7**	**result**
x86-64	–	`rdi`	`rsi`	`rdx`	`rcx`	`r8`	`r9`	–	`rax`
arm/EABI	–	`r0`	`r1`	`r2`	`r3`	stack	stack	stack	`r0-r3`
x86-64	`rax`	`rdi`	`rsi`	`rdx`	`r10`	`r8`	`r9`	–	`rax`
arm/EABI	`r7`	`r0`	`r1`	`r2`	`r3`	`r4`	`r5`	`r6`	`r0`

Position-Independent Gadget Semantics. Attackers need a whole set of gadgets to mount a successful attack. The final step of an exploit is often to call a security-sensitive function or a system call with attacker-specified arguments (*e.g.*, `execve` with "`/bin/sh`" as argument for a shell). The ABI-heterogeneity provided by DMON imposes another constraint on chaining gadgets to build such an exploit. Because different architectures have different calling conventions for system calls and subroutines, as shown in Table 6.1, the attacker should chain a sequence of gadgets that prepare the same set of arguments, but in a different way for each architecture.

For example, in an ARMv7 variant, the attacker must use `r7` to prepare a system call number, whereas in a x86-64 variant the same attacker must use `rax`. To show the difficulty of constructing a code-reuse attack that performs one or more system calls and/or subroutine calls, we analyzed the semantics of position-independent gadgets surviving under DMON. Specifically, we looked for gadgets that read a value from memory and write that value into the system call number register, or the registers for one of the first three arguments

of a system or function call. As shown in Figure 6.1, only a small fraction of the position-independent gadgets have suitable semantics for argument preparation (see 3rd to 6th bars in the figure). More interestingly, system call number preparation gadgets are rare compared to other argument preparation gadgets. In a standalone ARMv7 binary of Nginx, Redis, and ProFTPD, we could not find a single partial-pointer-overwrite based position-independent gadget which can load a system call number. Obviously then, we also could not find such gadgets among those that survive across architectures.

6.2.2 Structure Layout Diversity

Apart from code layout diversity we achieve from ISA-heterogeneity, DMON naturally provides data structure layout diversity. Due to differences in sizes of pointers and primitive data types, as well as differences in struct packing and alignment, data structures rarely have the same sizes and layouts across platforms. Diversifying structure layouts greatly raises the bar for attacks that require knowledge about data structure definitions including certain types of data-only attacks that rely on deterministic placement of structure fields [36, 24].

Previous NVX systems could achieve structure layout diversity by artificially reorganizing structures at compile time. However, in practice, only a limited number of structs can be diversified at compile time. Specifically, it is not safe to diversify (i) structures used as arguments or return types of external library functions, (ii) structures with an initialization list, (iii) structs cast to different types, etc. [45, 16]. We implemented existing type-based

Table 6.2: The number of diversified data structures.

	Artificial	DMON	total
Nginx 1.14.2	53	335	365
Lighttpd 1.4.52	15	95	116
Redis 5.0.1	57	158	209
ProFTPD 1.3.0	23	72	84

structure layout randomization techniques [45, 16], and we examined struct layouts in a set of server applications to show how much structure layout diversity DMON can naturally achieve, compared to the number of structures that can be artificially diversified. As shown in Table 6.2, our heterogeneous NVX system provides a much higher degree of structure layout diversity than one can achieve using a compiler-based technique.

Case Study: ProFTPD SSL Private Key Leak. Hu et al. demonstrated an information disclosure attack on ProFTPD, in which the attacker locates a base pointer to an SSL context data structure, and then uses Data-Oriented Programming (DOP) gadgets to traverse through the context and 6 other data structures, ultimately reaching a private key, which is then leaked to a remote attacker [36]. DMON can prevent this attack because the layouts of the 6 data structures differ across architectures. We examined the relevant data structures in ARMv7 and x86-64 binaries of ProFTPD and found that 4 of the 6 pointer fields that need to be dereferenced in this attack are located at different offsets in the two binaries. A DOP exploit that traverses through the structs therefore cannot simultaneously reach and leak the private key on both platforms without triggering a divergence in DMON.

Chapter 7

Performance Evaluation

We conducted an extensive performance evaluation of DMON using handwritten microbenchmarks (see Chapter 7.1), as well as popular high-performance server applications (see Chapter 7.2). We tried two different configurations:

The low-end configuration had an ARMv8 variant running on a Raspberry Pi 3 Model Bboard with a quad-core 1.2GHz Broadcom BCM2837 64-bit CPU and 1GB of RAM, running the 64-bit ARM Debian 9 distribution of GNU/Linux, as well as an x86-64 variant running on a desktop machine with a quad-core Intel i5-6500 CPU and 16GB of RAM, running the x86-64 version of Ubuntu 16.04.5 LTS. The machines were connected through a *private* 100 megabit Ethernet connection with approximately 0.5 ms latency. Note that we used a Raspberry Pi 3, since we did not have access to a higher-end ARM machine at the time of writing.

The high-end configuration had an x86-64 variant running on a desktop machine with an octa-core Intel i9-9900K CPU and 32GB of RAM, and an x86-64 variant running on a machine with a quad-core Intel i5-6500 CPU and 16GB of RAM. Both machines ran the x86-

64 version of Ubuntu 16.04.5 LTS and were connected using a private 100 gigabit connection between two Mellanox ConnectX Ethernet interface cards. These RDMA-capable cards support the Mellanox Messaging Accelerator, a user space networking library with TCP support and sub-microsecond latency. In both configurations, we ran the leader variant on the slower machine. This choice minimizes the time the leader variant has to wait for the follower variant to send the system call state information (see Chapter 3.1).

We evaluated two implementations of RC-COM (see Chapter 5.1) for the low-end configuration. The first implementation, which appears as KTCP (short for kernel space TCP) in the graphs, uses the standard Linux TCP/IP stack. The second implementation uses the ENet protocol. For the high-end configuration, we additionally evaluated an implementation that leverages the Mellanox Messaging Accelerator library. This implementation appears as UTCP (short for user space TCP) in the graphs. We could not test this UTCP implementation for low-end configuration as we could not find the appropriate hardware for our ARMv8 board. Finally, we evaluated the impact of our replication and cross-checking optimizations described in Chapter 5.2. Our Asynchronous Cross-Checking and Permissive Filesystem Access optimizations appear as ACC and PFA respectively in the graphs.

7.1 Microbenchmarks

To measure the overhead introduced by DMON, we designed microbenchmarks to test different combinations of cross-checking and replication strategies. The microbenchmarks execute the same system call ten million times.

We used the following system calls:

1. **sys_read(STATIC_FILE_FD, buf, 512)** is treated as a moderately *sensitive* system call. As such, this microbenchmark benefits from our asynchronous cross-checking

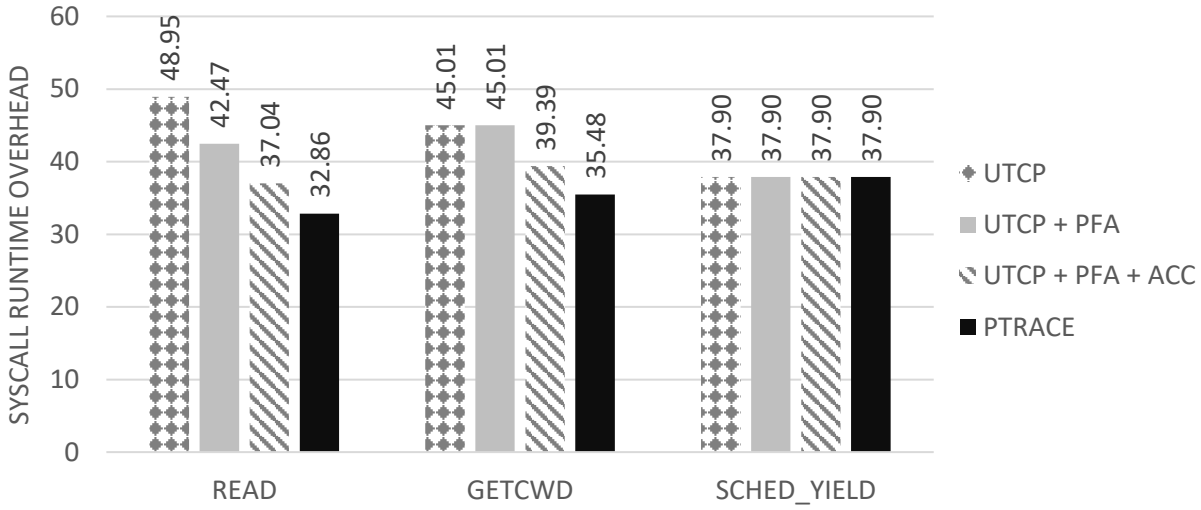
optimization (see Chapter 5.2). Since the file we are reading is *static*, DMON also skips replication if the permissive file system access is enabled.

2. **sys_getcwd(buf, 512)** The results of this system call do not need to be replicated, as long as the current working directory is either the application's root directory, or one of its subdirectories (see Chapter 5.2).

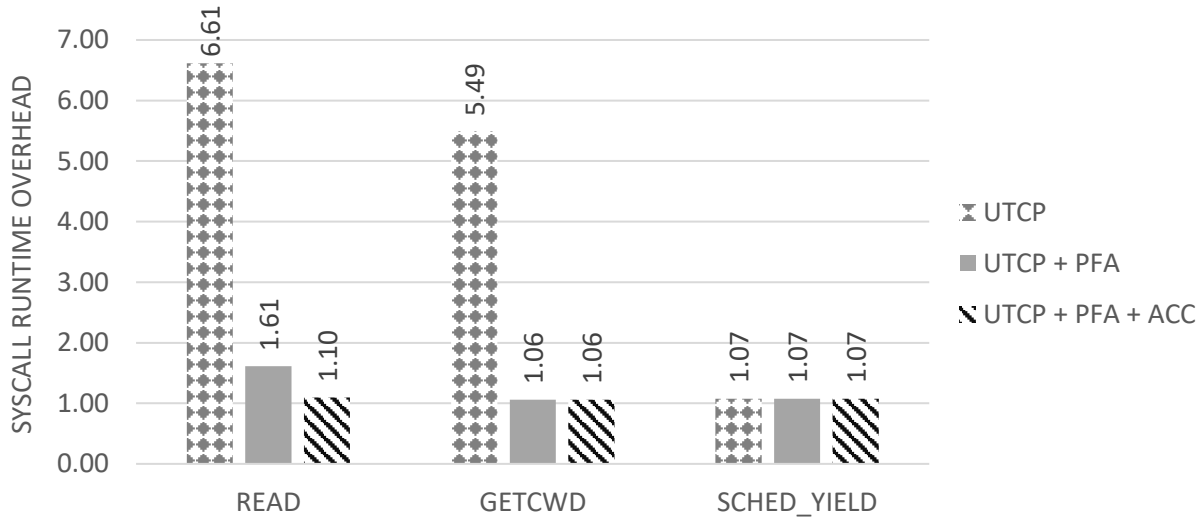
3. **sys_sched_yield()** is a representative of system calls that require neither cross-checking nor replication.

For each microbenchmark, we measured the execution time under DMON's high-end configuration relative to the native execution time *on the slowest machine*. Figure 7.1(a) shows the mathematical average of the run-time for three runs of each benchmark, relative to the native execution time. We used our UTCP implementation of RC-COM for all experiments, but did run separate tests with and without our permissive file access (PFA) and asynchronous cross-checking (ACC) optimizations. We also measured the execution time without cross-checking and replication (PTRACE). This experiment shows that the `ptrace` API is the main performance bottleneck in our system. Prior work illustrates that replacing `ptrace`-based monitoring by in-process alternatives allows for a much wider range of security-performance trade-offs [78, 40].

The results show that most of the overhead can be attributed to two sources: the network communication of our replication and cross-checking mechanisms, and the context switching caused by `ptrace`. PFA reduces the overhead of our `read` benchmark from 48.95× to 42.47×, but does not affect the `getcwd` and `sched_yield` benchmarks. This is unsurprising since `read` is the only of the three system calls that accesses *static* files. ACC further decreases overhead of `read` and `getcwd`, from 42.47× to 37.04× and from 45.01× to 39.39× respectively. `sched_yield`'s performance is unaffected, since DMON does not perform any cross-checking for this system call. Finally, the rightmost columns in Figure 7.1(a) indicate



(a) Fully distributed ptrace-based NVX



(b) Proof of concept distributed in-process NVX

Figure 7.1: Microbenchmarks for *high-end* configuration.

that the context switching overhead of `ptrace` is by far the biggest contributor to DMON's overhead.

We hypothesized that monitoring *non-sensitive* system calls in-process, as was done in prior work [34, 78, 58], would substantially reduce the context switching overhead, and set up an experiment to confirm this hypothesis. Specifically, we implemented a minimal prototype of a distributed in-process NVX system using the `syscall_intercept` [68], and evaluated

it on the same microbenchmarks we used for the `ptrace`-based prototype. Our in-process prototype implements the optimizations described in Chapter 5.2, but only supports a small set of system calls. Figure 7.1(b) shows that in-process monitoring drastically reduces the per-system call overhead from 32.86-37.90× to only 6-10% with all optimizations enabled. Note that `syscall_intercept` can be replaced by other system call interception mechanisms. MVArmor [40], which is one of the fastest NVX systems in the literature, uses a modified version of Dune [7] to intercept system calls. We expect that adapting the same mechanism would enhance the performance of our prototype.

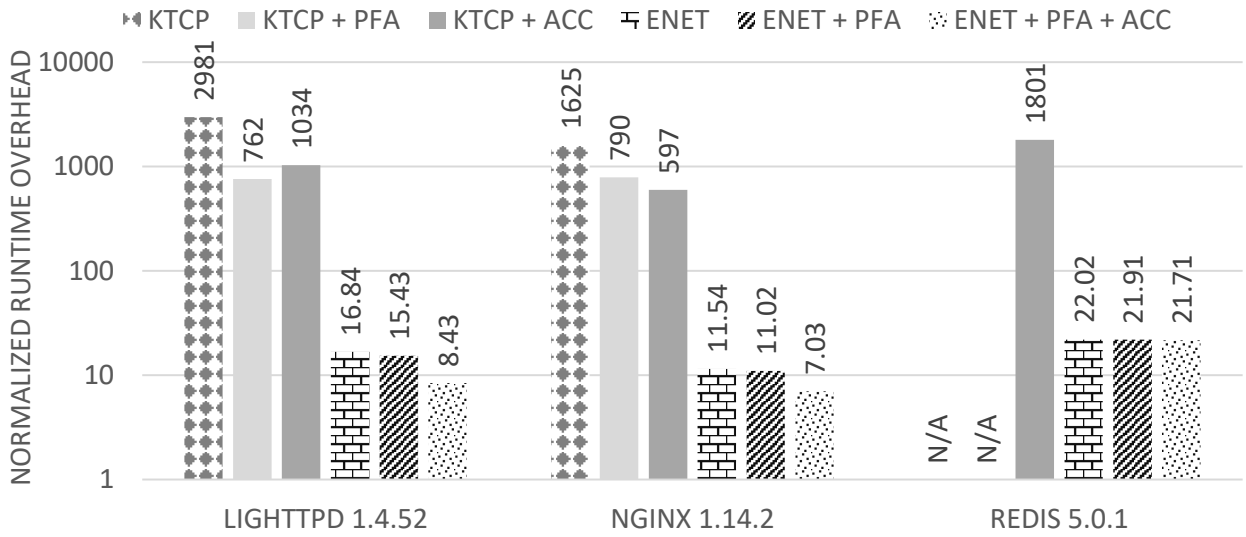
7.2 Server Benchmarks

We evaluated DMON on 3 popular server applications — `Nginx 1.14.2`, `Lighttpd 1.4.52` and `Redis 5.0.1` — that were also used to evaluate prior work [34, 78, 40, 46]. For each of our experiments, we connected a benchmarking client to the leader machine through a 100 megabit Ethernet connection (for our low-end configuration) or a 1 gigabit Ethernet connection (for the high-end configuration). The performance evaluation reported in this chapter considers the overhead in terms of effect on throughput. Figure 7.2 shows our results.

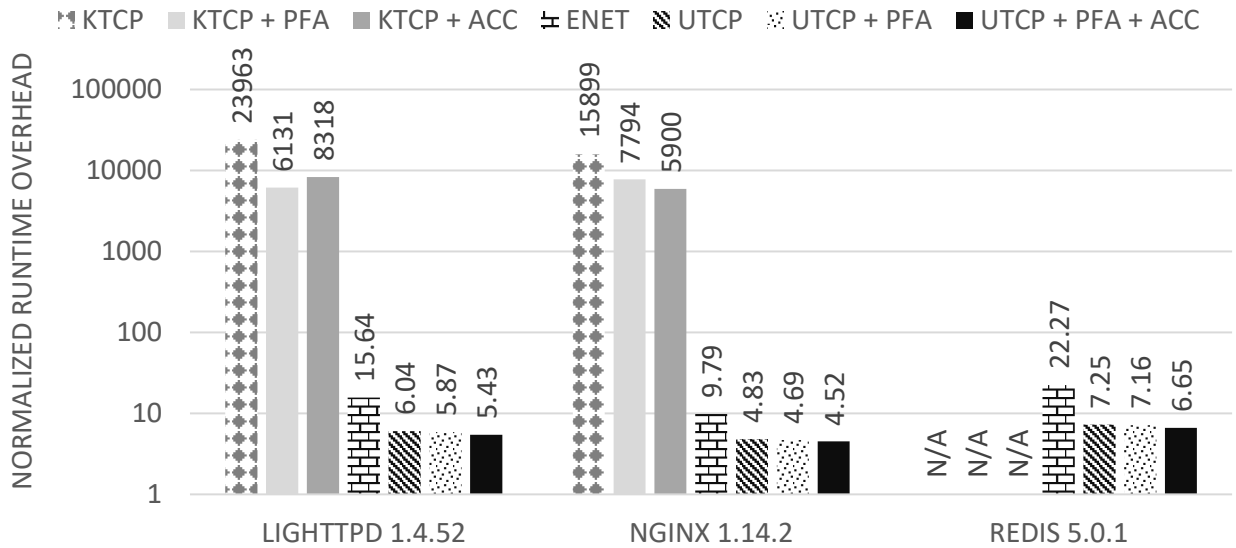
We used the `wrk` client to evaluate `Nginx` and `Lighttpd`, and the `redis-benchmark` utility to evaluate `Redis`. We configured `wrk` to repeatedly request the same static 4KB web page for 10 seconds using 10 parallel connections, and `redis-benchmark` to simulate 50 clients issuing 100,000 requests in total. Running `redis-benchmark` under DMON's slowest configurations would take over a day, so we skipped them and denote it as *N/A* in Figure 7.2.

We used the network link between the benchmark client machine and the machine that runs the leader variant as-is. This meant that the latency on the 100 megabit link was just under 0.5 ms, whereas the latency on the 1 gigabit link was under 0.1 ms.

The benchmarking client was able to fully saturate the leader variant's machine in both cases. With all of DMON's optimizations enabled, the performance overheads ranged between 7.03× and 21.71× for the low-end configuration, and between 4.52× and 6.65× for the high-end configuration. Note that DMON's overheads are lower than a traditional `ptrace`-based NVX system on the same hardware (see Chapter 7.3).



(a) Low-end configuration



(b) High-end configuration

Figure 7.2: Server benchmarks in two configurations.

7.3 Comparison With Other NVX Systems

We compared the performance of DMON with traditional NVX systems. Thanks to our inter-monitor communication optimizations, DMON achieves similar (or better) performance than traditional single-host NVX systems with `ptrace`-based monitors. Specifically, GHUMVEE (the state-of-the-art `ptrace`-based NVX system) was tested on the same server applications (albeit slightly older versions), and in highly similar conditions, with a 1 gigabit link that had less than 0.1 ms of latency. GHUMVEE's overhead on `Lighttpd` was 7.0× on a saturated server (vs 5.43× for DMON), and 12.48× for `Redis` (vs 6.65× for DMON) [78]. Delegating the monitoring of *non-sensitive* system calls to an in-process monitor would substantially reduce the overhead, as was shown in prior work [34, 78, 58], as well as in Chapter 7.1.

We summarize our findings in Table 7.1. DMON (IP) refers to our PoC distributed in-process prototype and DMON (CP) refers to our distributed `ptrace`-based implementation. As GHUMVEE was not evaluated on microbenchmarks, and DMON (IP) currently does not support server applications, these numbers are shown as *N/A* in the table.

Table 7.1: Comparison with other NVX systems.

NVX System	Monitor Type	Distributed	Overhead	
			System Call	Server Apps
GHUMVEE[78]	CP	No	N/A	7.0-12.48×
DMON (CP)	CP	Yes	**32.86-37.90×**	**4.52-6.65×**
Varan[34]	IP	No	36-139%	11-37%
DMON (IP)	IP	Yes	**6-61%**	**N/A**

Chapter 8

Secure and Efficient Distributed Monitoring and Replication

We implemented DMON [80] as a distributed cross-process NVX system, since we expected network communication to be the dominant factor of performance degradation. However, in Chapter 7.1 we showed that context switching caused by `ptrace` is by far the biggest contributor to DMON's overhead, when we use suitable communication protocols and network adapters. Consequently, we can build an efficient, secure, distributed NVX system by replacing `ptrace` with a faster system call interception mechanism. In this chapter, we present DMON++, a *new*, distributed, hybrid NVX system, inspired from previous work [78, 80]. We show the challenges of building such a system, provide solutions to them and also evaluate its performance.

8.1 Challenges

8.1.1 Security

Our goal is to build a secure, efficient, distributed NVX system. Consequently, system call interception techniques used in Varan [34] or user space libraries such as `syscall_intercept` [68] are obvious candidates to replace `ptrace`. However, both of them are based on binary rewriting, which cannot ensure that all system calls are intercepted and monitored. For example, a careful attacker could execute an unaligned *syscall* instruction, avoiding monitoring. Furthermore, Varan's code is not open source and `syscall_intercept` supports only x86 architecture. However, a hybrid design similar to ReMon could be potentially adapted and used as the basis of a secure, efficient distributed NVX system.

8.1.2 Distributed In-Process Monitoring and Replication

A hybrid NVX system consists of a cross-process monitor component (CP-MON) and an in-process monitor component (IP-MON) (see Chapter 2.2). Unlike a centralized in-process monitor, a distributed in-process monitor has to issue *networking-related* system calls for monitoring (called also cross-checking) and replication. Unfortunately, these system calls would be eventually intercepted and cross-checked by our security-oriented NVX system, causing observable divergences due to the asymmetry of the sending and receiving part of the inter-monitor communication. Consequently, the challenge to building a distributed in-process monitor is how to avoid intercepting these system calls, while monitoring the ones that are issued by the variants directly.

A naive approach is to add two *special* system calls to enable/disable system call monitoring. The main idea is to disable system call monitoring before we use the network channel for

inter-monitor communication, avoiding *undesired* monitoring that leads to false alarms, and re-enable it again when the data exchange is finished. Note that the two *special* system calls are not monitored and should never be called by the variant itself. This solution adds at least eight additional mode switches and *expensive* network communication compared to ReMon[1], making this design unsuitable for an efficient in-process monitor.

We could potentially do some static and/or dynamic analysis of the networking libraries to white-list *network-related* system calls used by the communication channel. However, this could be really challenging for Mellanox's Messaging Accelerator (VMA) [44], our fastest reliable communication channel, since this library is too complicated (thousands system calls are issued for exchange of just a dozen packets) and additional internal threads are used for packet handling, re-ordering and clock synchronization.

8.2 Design and Implementation

In this chapter we present DMON++, a distributed, efficient, security-oriented, hybrid NVX system. Our approach combines an existing distributed, cross-process NVX system (DCP-MON) [80] with a distributed in-process component (DIP-MON) to execute *non-sensitive* system calls efficiently

DMON++ supervises the execution of program variants that run in parallel in different physical machines. Its main goals are (i) to cross-check all of the security *sensitive* system calls issued by the variants, (ii) to execute *sensitive* system calls in lockstep and (iii) to disable cross-checking and lockstepping for security *non-sensitive* system calls, minimizing the overhead of our NVX system for these calls. We refer to previous work [78, 40] for different monitoring relaxation policies of system calls. DMON++ uses four main components to

[1]Each disable/re-enable sequence adds four mode switches. We need to disable and re-enable system call monitoring before both cross-checking and replication handlers.

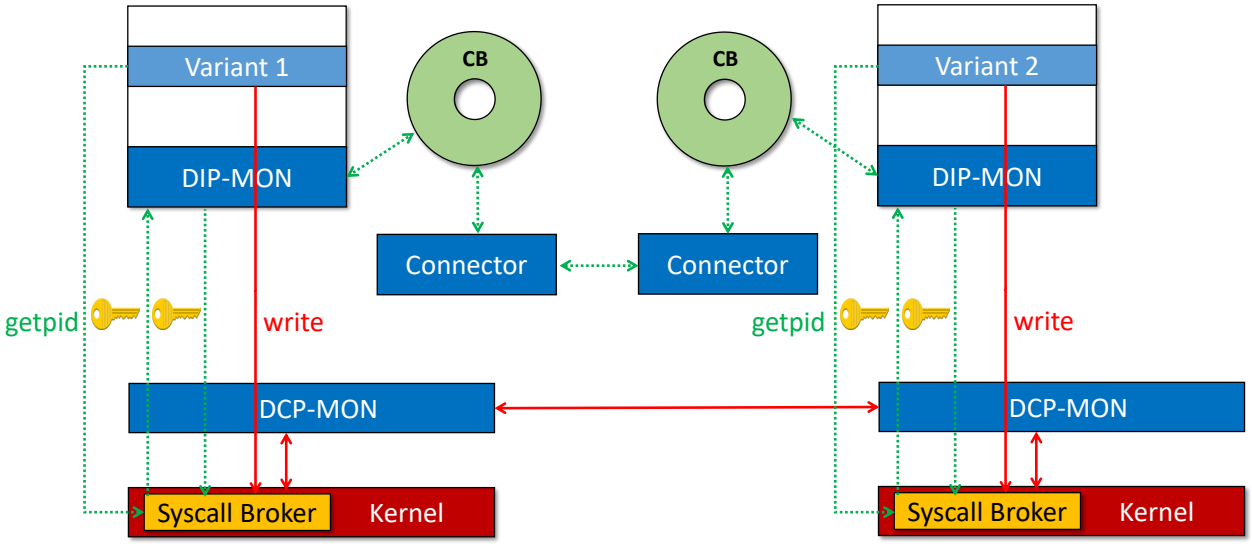


Figure 8.1: DMON++'s main components and interactions.

reach these goals:

1. **DMON** A security-oriented, distributed, cross-process monitor (DCP-MON) as discussed in Chapters 3, 4, 5, 6 and 7. DMON is responsible to handle all *sensitive* system calls when used as part of DMON++.

2. **DIP-MON** A distributed, in-process monitor loaded into each variant as a shared library. *Non-sensitive* system calls are forwarded to DIP-MON for cross-checking and replication. We show later how this component cooperates with Connector to provide this functionality.

3. **Syscall Broker** A small in-kernel component similar to the one used in ReMon. Syscall broker forwards *sensitive* system calls to DMON and *non-sensitive* ones to DIP-MON. In addition, it also restricts DIP-MON to execute only authenticated system calls.

4. **Connector** Each variant is also assigned a Connector component. Connector communicates with DIP-MON through a *special* shared memory segment, called communication buffer (CB), and is responsible for transferring data to/from Connector

components of the other variants.

These four components interact with each other whenever a variant executes a system call, as shown in Figure 8.1.

Our syscall broker intercepts system calls and decides if they will be forwarded to DMON or to DIP-MON depending on their security-sensitivity. DMON handles system calls as we described in previous chapters. `ptrace` is used to forward system calls to DMON, while our syscall broker is responsible to overwrite the program counter and forward *non-sensitive* system calls to DIP-MON. DMON++'s applies security restrictions to DIP-MON by using mechanisms similar to the ones described in Chapter 2.2.

In Chapter 8.1.2, we described challenges of building a distributed in-process monitor like DIP-MON. We tackle these challenges by introducing a Connector component and a *special* shared memory segment called communication buffer (CB). DIP-MON and Connector components are communicating using CB. DIP-MON is responsible to inform its corresponding Connector component when it needs to send/receive data for cross-checking or replication. On the other hand, Connector components exchange data through network and are implemented as separate processes. Consequently, the system calls issued for network communication are not visible from DIP-MON or DMON.

CB is a critical component of DMON++'s performance and is accessed by both DIP-MON and Connector. We need to ensure that no race conditions happen between DIP-MON and Connector components, and that memory read/writes happen in the correct order. To solve these problems, we implemented portable synchronization primitives using C++ atomic operations library.

8.2.1 The DIP-MON File Map

Both DMON and DIP-MON keep metadata of the open file descriptors in order to ensure I/O transparency and optimize cross-checking and replication handlers. However, both DMON and DIP-MON could open file descriptors (depending on the relaxation policy and the executed system call), leading to discrepancies of their metadata.

Each variant can map a read-only copy of this metadata in its address space. We refer to this metadata as DIP-MON file map and we implemented it in a way similar to CB. DIP-MON file map is shared between DMON and DIP-MON ensuring that both of them have a *complete* view of the variant's file descriptors at any moment. A similar mechanism was also implemented in ReMon [78]. To avoid re-ordering of read/writes and race conditions, we implemented portable synchronization primitives using C++ atomic operations library.

8.2.2 Securing the Design

To secure our design, we mimic techniques described in Chapter 2.2. System calls that could tamper CB or DIP-MON file map are always handled by DMON. In addition, we ensure that sensitive values like the pointers to CB or DIP-MON file map are not leaked to the stack by using fixed registers. Finally, to fully hide the location of CB, while still allowing benign accesses, we ensure that the pointer to the CB is only stored in kernel memory and passed to DIP-MON through a fixed register *only* when needed.

8.3 Optimizations

DMON++ is a distributed, hybrid NVX system aiming for both security and performance. DMON's optimizations, as described in Chapter 5.2, are *inherited* also in DMON++. Fur-

thermore, we implemented an additional set of optimizations for DIP-MON, the in-process part of DMON++.

Selective Cross-Checking DMON++ forwards *non-sensitive* system calls to DIP-MON. ReMon [79] and MvArmor [40] do not cross-check the system call arguments of these calls to minimize overhead. We implemented this technique in DIP-MON and we refer to it as *selective cross-checking*.

8.3.1 Extended Selective Replication

Similar to DMON, DMON++ uses network for replicating system call results. Consequently, avoiding replication when possible is of great importance. We refer to all these optimizations as *extended selective replication*. Extended selective replication includes all the optimizations described in Chapter 5.2.2 and extends them.

Asynchronous Replication Connector is implemented as a standalone process. Consequently, the DIP-MON assigned to the leader variant could just copy the leader's system call result to CB, inform Connector and return. This avoids stalling the leader variant, until the network communication between the Connector components is finished. This option is always enabled in DIP-MON.

Relaxed Permissive Filesystem Access We discovered that our Permissive Filesystem Access (PFA) optimization described in Chapter 5.2.2 was too conservative. We extend PFA to all files that are located in application's sub-directories. We assume that application files are in the same paths on both physical machines. We refer to these files again as *static* files. Consequently, each variant could perform I/O operations (including writes) in its own copy

of the file. We assume that these files are not changed by a source other than the application itself, ensuring consistency and identical inputs. In case that a monitored application tries to create a file, one separate copy is created for each physical machine.

sys_setsockopt, sys_open and similar system calls. Avoiding replication for these system calls is important (from a performance perspective), since they are executed frequently in popular server applications. `sys_open` and similar [3] system calls open and possibly create a file specified by *pathname*. In case of success, it returns an integer, representing the file descriptor value that corresponds to the *newly* opened file. NVX systems always need to replicate the results of `sys_open` and similar system calls, since leader and follower variants have different *opened* file descriptors, *e.g.*, network sockets are only open in the leader variant; otherwise variants would get different inputs in subsequent system calls, resulting in false alarms. DMON++ keeps metadata of opened file descriptors and their types. Since we know how the Linux kernel works [2], we could predict the file descriptor value returned to the leader variant, eliminating the need to replicate the results of `sys_open` and similar system calls.

Another system call that is always replicated by NVX systems is `sys_setsockopt` [4]. This system call sets options on the socket, passed as file descriptor, and returns 0 in case of success. On error, -1 is returned, and *errno* is set appropriately. DMON++ keeps metadata for all open sockets. Consequently, DMON++ could predict, with high confidence, the success of `sys_setsockopt`, by checking (using its metadata) the validity of the socket and the socket options, eliminating the need to replicate the results. Note that a "wrong" prediction may lead to inconsistencies, since variants will get different system call results. In the future, we want to provide solutions for this problem, *e.g.*, restarting these system calls in case of failure.

[2]The returned file descriptor is the minimum integer value that is available for a file descriptor and it depends on the previously opened file descriptors including files, network sockets and pipes.

8.4 Performance Evaluation

We evaluated the performance of DMON++ using the same handwritten microbenchmarks described in Chapter 7.1, as well as `Lighttpd`, a popular server application. Furthermore, we used the high-end configuration described in Chapter 7.

The Connector component leverages the Mellanox Messaging Accelerator library, to build a fast, ultra-low latency, reliable communication channel. In addition, we evaluated the impact of our replication and cross-checking optimizations described in Chapter 8.3. Our Selective Cross-Checking and Extended Selective Replication optimizations appear as SCC and ESR respectively in the graphs.

8.4.1 Microbenchmarks

We evaluated DMON++ on the same microbenchmarks that we used for DMON's evaluation (see Chapter 7.1). In addition, we tested the impact of our cross-checking and replication optimizations. Each system call was executed ten million times.

We used the following system calls:

1. **sys_read(STATIC_FILE_FD, buf, 512)** is treated as a moderately *sensitive* system call. As such, this microbenchmark benefits from our selective cross-checking optimization (see Chapter 8.3). Since the file we are reading is *static*, DMON++ also skips replication if the extended permissive file system access is enabled.

2. **sys_getcwd(buf, 512)** The results of this system call do not need to be replicated, as long as the current working directory is either the application's root directory, or one of its subdirectories (see Chapter 8.3).

3. **sys_sched_yield()** is a representative of system calls that require neither cross-checking nor replication.

For each microbenchmark, we measured the execution time under DMON++ relative to the native execution time. Figure 8.2 shows the mathematical average of the run-time for three runs of each benchmark, relative to the native execution time. We did run separate tests with and without our extended selective replication (ESR) and selective cross-checking (SCC) optimizations

The results show that most of the overhead can be attributed to the context switching caused by `ptrace` and to the network communication of our replication and cross-checking handlers. The leftmost columns in Figure 8.2 show the overhead when DIP-MON is not used and DMON is configured with all its optimizations. Enabling DIP-MON drastically reduces DMON++'s overhead for all microbenchmarks; from 37.04× to 15.66× for `read`, from 39.39× to 9.70× for `getcwd` and from 37.90× to 2.87× for `sched_yield`. ESR reduces the overhead of our `read` benchmark from 15.66× to 12.24×, but does not affect the `getcwd` and `sched_yield` benchmarks. This is unsurprising since `read` is the only of the three system calls that accesses *static* files. SCC decreases the overhead of `read` and `getcwd`, from 15.66× to 7.58× and from 9.70× to 2.79× respectively. `sched_yield`'s performance is unaffected, since DMON++ does not perform any cross-checking for this system call. Finally, the rightmost columns in Figure 8.2 show the execution time when all optimizations are enabled.

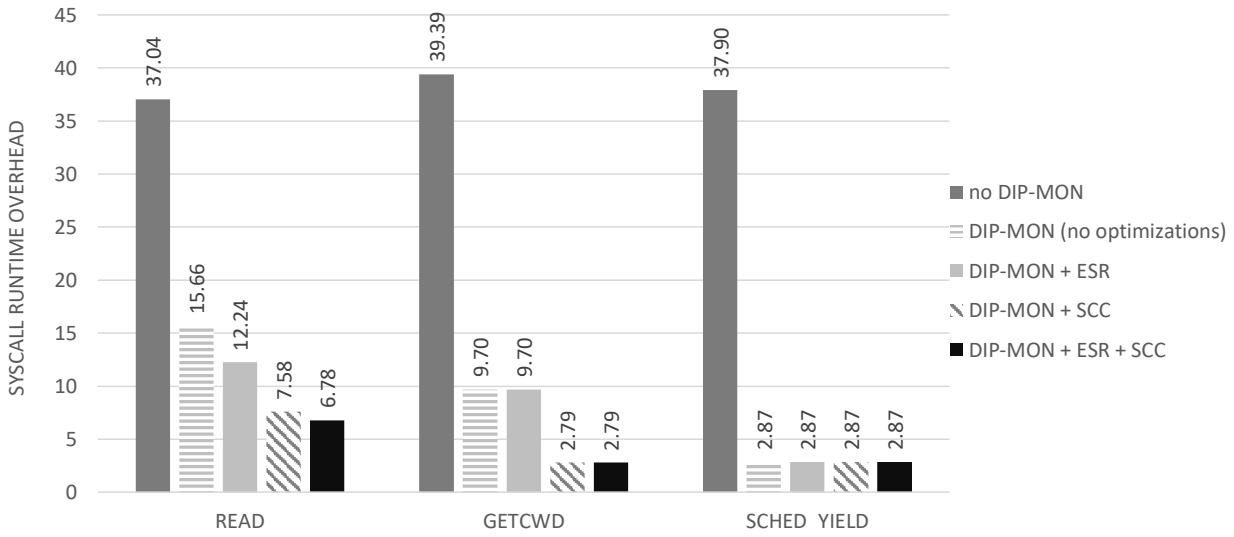


Figure 8.2: Microbenchmarks for DMON++.

8.4.2 Evaluating DMON++ on Lighttpd

We evaluated DMON++ on a popular server application — `Lighttpd 1.4.52` — that was also used to evaluate DMON. We used exactly the same benchmarking tools and settings that are described in Chapter 7.2. We measured the overhead under DMON++ relative to the native execution time. The performance evaluation reported in this chapter considers the overhead in terms of throughput. Figure 8.3 shows our results.

The leftmost columns in Figure 8.3 show the overhead when DIP-MON is not used and DMON is configured with all its optimizations. Enabling DIP-MON drastically reduces DMON++'s overhead from 5.43× to 1.62×. ESR and SCC reduce the overhead even more to 1.54× and 1.072× respectively. Finally, the rightmost columns in Figure 8.3 show the run-time overhead when all optimizations are enabled. With all of our optimizations active, `Lighttpd`'s overhead, when running under DMON++, drops from 5.43× to just 3.2%.

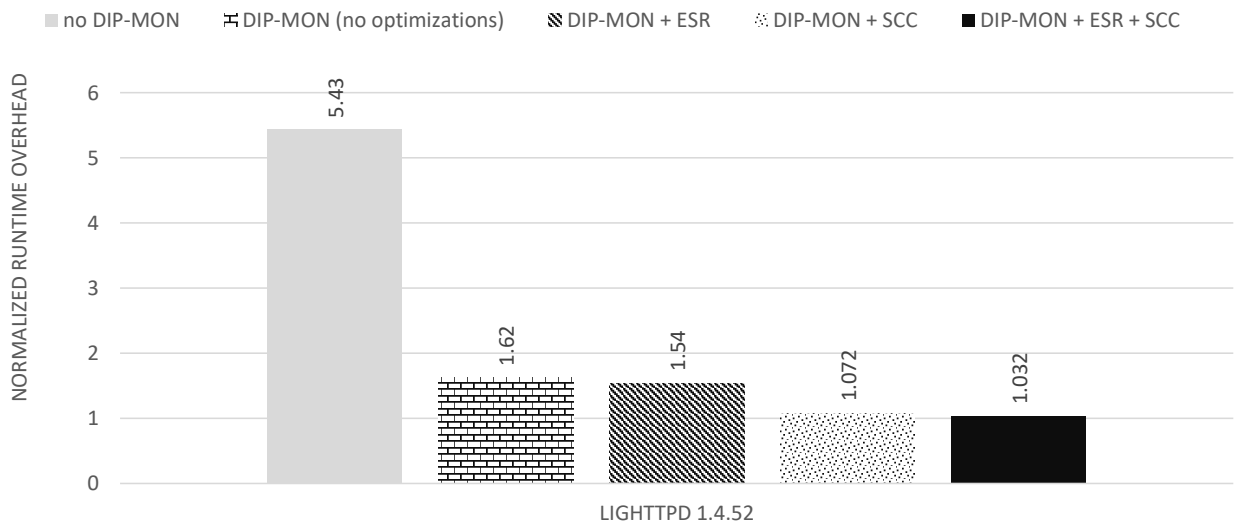


Figure 8.3: Lighttpd benchmark for DMON++.

Chapter 9

Discussion, Related Work and Conclusion

"UNIX is *simple*. It just takes a `genius` to understand its *simplicity*."

– *Dennis Ritchie*

9.1 Discussion

Additional Experiments. High-end ARM servers are available in the market. These machines are compatible with our optimized networking hardware [48] and Mellanox Messaging Accelerator user space networking library [44] supports ARM platforms. We plan to purchase an ARM server and repeat the experiments described in Chapter 8.4. We expect this to require minimal to none engineering effort, since both DIP-MON's and DMON's code are portable. Finally, we also want to experiment with more server applications.

Performance Improvements. While developing DMON and DMON++ we experimented with alternative implementations for the network communication channel. In the future, we could also leverage the SocketXtreme API in the Mellanox Messaging Accelerator library to speed up network operations. This would reduce the latency and increase the bandwidth on our inter-monitor network links, thus improving cross-checking and replication performance. We did not use the SocketXtreme API in our current prototype as it is still under active development. Currently, the API has non-trivial limitations, and it is not available for all platforms.

Leveraging Hardware Features. A potential advantage of running variants on different architectures is that the NVX system could leverage hardware security features available on one platform to protect software running on other platforms. A future revision of the ARMv8 architecture (ARMv8.5-A), for example, will include a feature called Memory Tagging. This feature allows the program to add tags to every pointer and every memory allocation. When the program dereferences a pointer, the CPU automatically checks if the pointer tag matches the allocation tag. Memory tagging, if properly implemented, can detect both spatial and temporal memory errors. A hypothetical configuration in which DMON runs one variant on an ARMv8.5-A CPU and one variant on an Intel x86-64 CPU could be used to bring the benefits of memory tagging to Intel x86-64 software. In a similar manner, DMON could enable Intel's Control-flow Enforcement Technology (CET) for ARM programs, and ARMv8.3's pointer authentication for Intel x86-64 programs.

Micro-Architectural Attacks. While our primary focus was on defending against memory exploits, we believe DMON might also be able to stop certain micro-architectural attacks. Rowhammer attacks in particular would become exceedingly hard to launch against DMON [73, 64, 29]. Rowhammer attacks induce bit flips in so-called weak DRAM cells by rapidly and repeatedly accessing adjacent DRAM rows. To build reliable Rowhammer

attacks, the attacker needs to know exactly how the memory controller translates physical memory addresses into DRAM addresses [57, 70]. Translation schemes differ greatly across platforms, however, which makes Rowhammer attack payloads non-portable. Moreover, even if two machines did have memory controllers using the same translation scheme, Rowhammer attacks would still fail with high probability under DMON, as the positions of the weak cells tend to differ for any given pair of DRAM modules.

Efficient Hardware-assisted Isolation. Memory protection keys (MPK) is a hardware feature recently added to x86 that allows protection domain switches in user space. Researchers already use protection keys for efficient, secure in-process isolation of libraries and security-critical parts of the applications [30, 71, 56]. We could potentially use the same idea to isolate the in-process monitor of an NVX system.

9.2 Related Work

N-Variant eXecution. The idea of running diversified software variants in parallel for increased security is not new. Inspired by Chen and Avizienis' seminal work on N-Version Programming [14, 6], Berger and Zorn proposed a system for probabilistic memory safety that could simultaneously execute identical variants with differently seeded randomizing memory allocators [8]. This system only supported applications that received input through `stdin` and that wrote output to `stdout`, however. Cox et al.'s N-Variant Systems monitored a much wider array of system calls and replicated I/O from various sources, thus supporting variants of non-trivial applications such as the Apache server [18]. Subsequent publications explored consistent delivery of asynchronous signals [10, 61], dealing with shared memory [10], thread synchronization [77], or address-dependent behavior [79], and new schemes for generating software variants [40, 76, 82, 46]. Other researchers suggested to use NVX systems for live

patch testing [47, 33, 39, 34, 42, 58]. Until recently NVX systems had to choose between security and performance. ReMon[78] was the first NVX system that was able to provide good performance without sacrificing security. Contrary to DMON [80], however, all of these systems, including ReMon, require that the variants are compiled for the same platform, and run on the same host.

Pina et al. proposed a Domain-Specific Language to specify expected divergences between different variants, and an NVX system that reconciles divergent variants [59]. Here, the assumption is that the variants have minor differences because they were not compiled from the same revision of the source code. DMON assumes that the variants were compiled from the same source code, but it does reconcile variants that diverge because of ISA and ABI-heterogeneity.

Heterogeneous-ISA Migration. Several researchers explored the idea of trans-ISA program migration. Devuyst et al. demonstrated performance and energy efficiency increases by migrating a program between the cores of a heterogeneous-ISA CPU [19]. Venkat et al. later showed that heterogeneous-ISA migration has potential security benefits [75]. These systems require specialized CPUs that are not widely available, whereas DMON runs on commodity hardware.

Replicated State Machines/Byzantine Fault Tolerance. Replicated State Machines (RSM) and Byzantine Fault Tolerance (BFT) techniques introduce high overheads. Researchers explored ways to make them practical [53, 55, 83, 41, 81] by tackling performance issues similar to a distributed NVX system. DMON's design could be strengthened by leveraging optimizations proposed in these fields.

9.3 Conclusion

To conclude, we identified issues in existing security-oriented NVX systems regarding their performance and security guarantees. We presented two *new* NVX systems ReMon and DMON, as solutions to the aforementioned problems. ReMon bridges the performance gap between reliability-oriented and security-oriented NVX systems. On the other hand DMON, which is the main focus of this dissertation, is a *novel* NVX system that is able to raise the bar for state-of-the-art attacks that are able to bypass existing NVX systems, including ReMon. However, DMON suffers from significant performance degradation. Finally, we present DMON++, a *new*, hybrid, NVX system, that is based on DMON, and provides efficient, secure, distributed monitoring and replication.

9.3.1 ReMon – A Hybrid NVX System

ReMon is a *new*, hybrid N-Variant Execution system that combines advantages of both cross-process and in-process monitoring/replication techniques. ReMon uses GHUMVEE [79] to monitor *sensitive* system calls, while offloading the handling of the *non-sensitive* ones to a fast in-process monitor, IP-MON. We also proposed different relaxation policies that classify system calls in *sensitive* and *non-sensitive*. Finally, we implemented techniques to secure our design.

9.3.2 DMON – A Distributed Heterogeneous NVX System

DMON is a *novel*, distributed N-Variant Execution system that leverages diversity in instruction set architectures and application binary interfaces to protect against memory corruption attacks. To bypass DMON, attackers must provide exploits that simultaneously work on two or more different platforms. DMON's heterogeneous platform setting naturally provides code

layout diversity which greatly raises the bar for successful code-reuse attacks, and it naturally provides a higher level of structure layout diversity than what existing compiler-based techniques can provide.

To avoid benign divergences caused by expected cross-platform differences, we propose Platform-Independent State Canonicalization, a technique that transforms system call states into platform-independent states. Also, we introduce new optimization strategies to alleviate performance issues that are unique to the distributed NVX setting. Our performance evaluation shows that the proposed optimizations, combined with an optimized network protocol, greatly reduce the performance overhead without sacrificing DMON's security guarantees. DMON is able to perform on par with centralized, cross-process NVX systems.

9.3.3 DMON++ – A Hybrid Distributed NVX System

DMON++ is a distributed, efficient, security-oriented NVX system. It is a hybrid NVX system that combines DMON with a distributed in-process monitor (DIP-MON). We show challenges of building a monitor like DIP-MON and present our solutions. We also propose a set of *new* optimizations. Our evaluation shows that DMON++ introduces only 3.2% overhead for Lighttpd.

Chapter 10

Contributions to Research Projects

The chapters included in this dissertation are based on two research project, ReMon and DMON, which are the collaborative work of researchers from University of California, KU Leuven, Ghent University, Immunant, Inc. and Seoul National University. DMON++ was implemented as an extension of DMON project. The below details my involvement in each of these works.

Chapter 2.2: ReMon – A Hybrid NVX System ReMon is joint work with Stijn Volckaert, who is the first author of the paper [78] and the main developer of this NVX system. My contribution consists of the elimination of all indirect jumps for IP-MON by inlining `glibc` functions needed for copying data from/to the replication buffer (RB).

DMON NVX System I am the main developer of DMON and the first author of the paper that is based on this system [80]. I performed all the work in designing and implementing the DMON NVX system and also evaluating it, with a few notable exceptions. My coauthor Dokyung Song did most of the work needed for the security analysis of DMON described in Chapter 6. My coauthor Yeoul Na implemented type-based structure layout

randomization and structure layout analysis described in Chapter 6.2.2. My coauthor Fabian Parzefall implemented the ENet-based communication channel described in Chapter 5.1. He is also responsible for several small improvements of DMON's build and bootstrap systems.

Bibliography

[1] ptrace(2) - Linux Manual Page.: `http://man7.org/linux/man-pages/man2/ptrace.2.html`.

[2] process_vm_read(2) - Linux Manual Page.: `http://man7.org/linux/man-pages/man2/process_vm_readv.2.html`.

[3] open(2) - Linux Manual Page.: `http://man7.org/linux/man-pages/man2/open.2.html`.

[4] setsockopt(2) - Linux Manual Page.: `https://linux.die.net/man/2/setsockopt`.

[5] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[6] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering (TSE)*, (12):1491–1501, 1985.

[7] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[8] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[9] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[10] D. Bruschi, L. Cavallaro, and A. Lanzi. Diversified process replicæ for defeating memory error exploits. In *IEEE Performance, Computing, and Communications Conference (IPCCC)*, 2007.

[11] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):16, 2017.

[12] L. Cavallaro. *Comprehensive Memory Error Protection via Diversity and Taint-Tracking*. PhD thesis, Universita Degli Studi Di Milano, 2007.

[13] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[14] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *International Symposium on Fault-Tolerant Computing (FTCS)*, 1978.

[15] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, 2005.

[16] Z. Chen and H. Han. Attack mitigation by data structure randomization. In *International Symposium on Foundations and Practice of Security*, pages 85–93. Springer, 2016.

[17] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[18] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *USENIX Security Symposium*, 2006.

[19] M. DeVuyst, A. Venkat, and D. M. Tullsen. Execution migration in a heterogeneous-isa chip multiprocessor. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[20] T. Durden. Bypassing PaX ASLR protection. *Phrack Magazine*, 11, 2002.

[21] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[22] M. Franz. Making multivariant programming practical and inexpensive. *IEEE Security and Privacy*, 16(3):90–94, May 2018.

[23] T. Garfinkel, B. Pfaff, M. Rosenblum, et al. Ostia: A delegating architecture for secure system call interposition. In *NDSS*, 2004.

[24] R. Gil, H. Okhravi, and H. Shrobe. There's a hole in the bottom of the C: On the effectiveness of allocation protection. In *IEEE Cybersecurity Development (SecDev)*, 2018.

[25] GNU C Library - `https://www.gnu.org/software/libc/`.

[26] E. Göktas, B. Kollenda, P. Koppe, E. Bosman, G. Portokalidis, T. Holz, H. Bos, and C. Giuffrida. Position-independent code reuse: On the effectiveness of ASLR in the absence of information disclosure. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018.

[27] ENet: Reliable UDP networking library. `http://enet.bespin.org`.

[28] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[29] D. Gruss, C. Maurice, and S. Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2016.

[30] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 489–504, Renton, WA, July 2019. USENIX Association.

[31] A. Homescu, T. Jackson, S. Crane, S. Brunthaler, P. Larsen, and M. Franz. Large-scale automated software diversityprogram evolution redux. *IEEE Trans. Dependable Secur. Comput.*, 14(2):158–171, Mar. 2017.

[32] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO 13, pages 1–11, USA, 2013. IEEE Computer Society.

[33] P. Hosek and C. Cadar. Safe software updates via multi-version execution. In *International Conference on Software Engineering (ICSE)*, 2013.

[34] P. Hosek and C. Cadar. Varan the unbelievable: An efficient n-version execution framework. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[35] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic generation of data-oriented exploits. In *USENIX Security Symposium*, 2015.

[36] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[37] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[38] M. Jonas, R. Michiel, and B. KD. Instrumenting jvms at the machine code level. in 3rd pa3ct-symposium, volume 19, 2003.

[39] D. Kim, Y. Kwon, W. N. Sumner, X. Zhang, and D. Xu. Dual execution for on the fly fine grained execution comparison. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[40] K. Koning, H. Bos, and C. Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2016.

[41] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 45–58, New York, NY, USA, 2007. ACM.

[42] Y. Kwon, D. Kim, W. N. Sumner, K. Kim, B. Saltaformaggio, X. Zhang, and D. Xu. LDX: Causality inference by lightweight dual execution. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[43] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[44] Mellanox's Messaging Accelerator(VMA): `https://github.com/Mellanox/libvma/`.

[45] Z. Lin, R. D. Riley, and D. Xu. Polymorphing software by randomizing data structure layout. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 107–126. Springer, 2009.

[46] K. Lu, M. Xu, C. Song, T. Kim, and W. Lee. Stopping memory disclosures via diversification and replicated execution. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2018.

[47] M. Maurer and D. Brumley. TACHYON: Tandem execution for efficient live patch testing. In *USENIX Security Symposium*, 2012.

[48] Mellanox. ConnectX-5 EN Single/Dual-Port Adapter Supporting 100Gb/s Ethernet. `http://www.mellanox.com/page/products_dyn?product_family=260&mtag=connectx_5_en_card`.

[49] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz. Opaque control-flow integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[50] Musl Libc - `https://www.musl-libc.org/`.

[51] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[52] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: Compiler enforced temporal safety for C. In *International Symposium on Memory Management (ISMM)*, 2010.

[53] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 308–318, New York, NY, USA, 2008. ACM.

[54] MANUAL PAGES, O. pledge - restrict system operations. `http://www.openbsd.org/cgi-bin/man.cgi/OpenBSD-current/man2/pledge.2`.

[55] J. Oplinger and M. S. Lam. Enhancing software reliability with speculative threads. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 184–196, New York, NY, USA, 2002. ACM.

[56] T. Park, K. Dhondt, D. Gens, Y. Na, S. Volckaert, and M. Franz. Nojitsu: Locking down javascript engines. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[57] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. Drama: Exploiting dram addressing for cross-cpu attacks. In *USENIX Security Symposium*, 2016.

[58] L. Pina, A. Andronidis, M. Hicks, and C. Cadar. Mvedsua: Higher availability dynamic software updates via multi-version execution. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[59] L. Pina, D. Grumberg, A. Andronidis, and C. Cadar. A dsl approach to reconcile equivalent divergent program executions. In *USENIX Annual Technical Conference*, 2017.

[60] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM03, page 18, USA, 2003. USENIX Association.

[61] B. Salamat, T. Jackson, A. Gal, and M. Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *European Conference on Computer Systems (EuroSys)*, 2009.

[62] B. Salamat, T. Jackson, C. W. Gregor, Wagner, and M. Franz. Run-time defense against code injection attacks using replicated execution. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 8(4), 2011.

[63] S. Schirra. Ropper. `https://github.com/sashs/Ropper`, 2014.

[64] M. Seaborn and T. Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. In *BlackHat USA*, 2015.

[65] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[66] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[67] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. SoK: Sanitizing for security. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[68] System call intercepting library: `https://github.com/pmem/syscall_intercept`.

[69] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[70] A. Tatar, C. Giuffrida, H. Bos, and K. Razavi. Defeating software mitigations against rowhammer: A surgical precision hammer. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2018.

[71] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. {ERIM}: Secure, efficient in-process isolation with protection keys ({MPK}). In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1221–1238, 2019.

[72] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrida. The dynamics of innocent flesh on the bone: Code reuse ten years later. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[73] V. Van Der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[74] Linux Programmers Manual. vdso(7) - Linux Manual Page.

[75] A. Venkat, S. Shamasunder, H. Shacham, and D. M. Tullsen. Hipstr: Heterogeneous-isa program state relocation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[76] S. Volckaert, B. Coppens, and B. De Sutter. Cloning your gadgets: Complete ROP attack immunity with multi-variant execution. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 13(4):437–450, 2016.

[77] S. Volckaert, B. Coppens, B. De Sutter, K. De Bosschere, P. Larsen, and M. Franz. Taming parallelism in a multi-variant execution environment. In *European Conference on Computer Systems (EuroSys)*, 2017.

[78] S. Volckaert, B. Coppens, A. Voulimeneas, A. Homescu, P. Larsen, B. De Sutter, and M. Franz. Secure and efficient application monitoring and replication. In *USENIX Annual Technical Conference*, 2016.

[79] S. Volckaert, B. De Sutter, T. De Baets, and K. De Bosschere. GHUMVEE: efficient, effective, and flexible replication. In *International Symposium on Foundations and Practice of Security (FPS)*, 2012.

[80] A. Voulimeneas, D. Song, F. Parzefall, Y. Na, P. Larsen, M. Franz, and S. Volckaert. Distributed heterogeneous n-variant execution. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2020.

[81] B. Wester, J. Cowling, E. B. Nightingale, P. M. Chen, J. Flinn, and B. Liskov. Tolerating latency in replicated state machines through client speculation. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 245–260, Berkeley, CA, USA, 2009. USENIX Association.

[82] M. Xu, K. Lu, T. Kim, and W. Lee. Bunshin: compositing security mechanisms through diversification. In *USENIX Annual Technical Conference*, 2017.

[83] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient architectural support for software debugging. In *International Symposium on Computer Architecture (ISCA)*, 2004.

[84] S. sterlund, K. Koning, P. Olivier, A. Barbalace, H. Bos, and C. Giuffrida. kMVX: Detecting kernel information leaks with multi-variant execution. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

Appendix A

Inlining Example

A.1 Original Memcpy Routine

```
.text
.globl __memcpy_sse2_unaligned
.type __memcpy_sse2_unaligned, @function
__memcpy_sse2_unaligned:
.cfi_startproc
movq %rsi, %rax
leaq (%rdx,%rdx), %rcx
subq %rdi, %rax
subq %rdx, %rax
cmpq %rcx, %rax
jb .Loverlapping
cmpq $16, %rdx
jbe .Lless_16
movdqu (%rsi), %xmm8
cmpq $32, %rdx
movdqu %xmm8, (%rdi)
movdqu -16(%rsi,%rdx), %xmm8
movdqu %xmm8, -16(%rdi,%rdx)
ja .L31
.Lreturn:
movq %rdi, %rax
ret
.p2align 4,,10
.p2align 4
.L31:
movdqu 16(%rsi), %xmm8
cmpq $64, %rdx
movdqu %xmm8, 16(%rdi)
movdqu -32(%rsi,%rdx), %xmm8
```

```
movdqu %xmm8, -32(%rdi,%rdx)
jbe .Lreturn
movdqu 32(%rsi), %xmm8
cmpq $128, %rdx
movdqu %xmm8, 32(%rdi)
movdqu -48(%rsi,%rdx), %xmm8
movdqu %xmm8, -48(%rdi,%rdx)
movdqu 48(%rsi), %xmm8
movdqu %xmm8, 48(%rdi)
movdqu -64(%rsi,%rdx), %xmm8
movdqu %xmm8, -64(%rdi,%rdx)
jbe .Lreturn
leaq 64(%rdi), %rcx
addq %rdi, %rdx
andq $-64, %rdx
andq $-64, %rcx
movq %rcx, %rax
subq %rdi, %rax
addq %rax, %rsi
cmpq %rdx, %rcx
je .Lreturn
movq %rsi, %r10
subq %rcx, %r10
leaq 16(%r10), %r9
leaq 32(%r10), %r8
leaq 48(%r10), %rax
.p2align 4,,10
.p2align 4
.Lloop:
movdqu (%rcx,%r10), %xmm8
movdqa %xmm8, (%rcx)
movdqu (%rcx,%r9), %xmm8
movdqa %xmm8, 16(%rcx)
movdqu (%rcx,%r8), %xmm8
movdqa %xmm8, 32(%rcx)
movdqu (%rcx,%rax), %xmm8
movdqa %xmm8, 48(%rcx)
addq $64, %rcx
cmpq %rcx, %rdx
jne .Lloop
jmp .Lreturn
.Loverlapping:
cmpq %rsi, %rdi
jae .L3
testq %rdx, %rdx
.p2align 4,,5
je .Lreturn
movq %rdx, %r9
leaq 16(%rsi), %rcx
leaq 16(%rdi), %r8
shrq $4, %r9
movq %r9, %rax
salq $4, %rax
cmpq %rcx, %rdi
setae %cl
cmpq %r8, %rsi
```

```
setae %r8b
orl %r8d, %ecx
cmpq $15, %rdx
seta %r8b
testb %r8b, %cl
je .L16
testq %rax, %rax
je .L16
xorl %ecx, %ecx
xorl %r8d, %r8d
.L7:
movdqu (%rsi,%rcx), %xmm8
addq $1, %r8
movdqu %xmm8, (%rdi,%rcx)
addq $16, %rcx
cmpq %r8, %r9
ja .L7
cmpq %rax, %rdx
je .Lreturn
.L21:
movzbl (%rsi,%rax), %ecx
movb %cl, (%rdi,%rax)
addq $1, %rax
cmpq %rax, %rdx
ja .L21
jmp .Lreturn
.Lless_16:
testb $24, %dl
jne .Lbetween_9_16
testb $4, %dl
.p2align 4,,5
jne .Lbetween_5_8
testq %rdx, %rdx
.p2align 4,,2
je .Lreturn
movzbl (%rsi), %eax
testb $2, %dl
movb %al, (%rdi)
je .Lreturn
movzwl -2(%rsi,%rdx), %eax
movw %ax, -2(%rdi,%rdx)
jmp .Lreturn
.L3:
leaq -1(%rdx), %rax
.p2align 4,,10
.p2align 4
.L11:
movzbl (%rsi,%rax), %edx
movb %dl, (%rdi,%rax)
subq $1, %rax
jmp .L11
.Lbetween_9_16:
movq (%rsi), %rax
movq %rax, (%rdi)
movq -8(%rsi,%rdx), %rax
movq %rax, -8(%rdi,%rdx)
```

```
        jmp .Lreturn
        .L16:
        xorl %eax, %eax
        jmp .L21
        .Lbetween_5_8:
        movl (%rsi), %eax
        movl %eax, (%rdi)
        movl -4(%rsi,%rdx), %eax
        movl %eax, -4(%rdi,%rdx)
        jmp .Lreturn
        .cfi_endproc
        .size __memcpy_sse2_unaligned, .-__memcpy_sse2_unaligned
```

A.2 IP-MON's Memcpy Routine

```
STATIC INLINE void *ipmon_memcpy_ptr_ptr(void *destination, const void *source, size_t num)
{
        asm volatile("movq %[destination], %%rdi\n\t"
        "movq %[source], %%rsi\n\t"
        "movq %[num], %%rdx\n\t"
        "movq %%rsi, %%rax\n\t"
        "leaq (%%rdx,%%rdx), %%rcx\n\t"
        "subq %%rdi, %%rax\n\t"
        "subq %%rdx, %%rax\n\t"
        "cmpq %%rcx, %%rax\n\t"
        "jb 8f\n\t"
        "cmpq $16, %%rdx\n\t"
        "jbe 9f\n\t"
        "movdqu (%%rsi), %%xmm8\n\t"
        "cmpq $32, %%rdx\n\t"
        "movdqu %%xmm8, (%%rdi)\n\t"
        "movdqu -16(%%rsi,%%rdx), %%xmm8\n\t"
        "movdqu %%xmm8, -16(%%rdi,%%rdx)\n\t"
        "ja 10f\n\t"
        "11:\n\t"
        "movq %%rdi, %%rax\n\t"
        "jmp 20f\n\t"
        ".p2align 4,,10\n\t"
        ".p2align 4\n\t"
        "10:\n\t"
        "movdqu 16(%%rsi), %%xmm8\n\t"
        "cmpq $64, %%rdx\n\t"
        "movdqu %%xmm8, 16(%%rdi)\n\t"
        "movdqu -32(%%rsi,%%rdx), %%xmm8\n\t"
        "movdqu %%xmm8, -32(%%rdi,%%rdx)\n\t"
        "jbe 11b\n\t"
        "movdqu 32(%%rsi), %%xmm8\n\t"
        "cmpq $128, %%rdx\n\t"
        "movdqu %%xmm8, 32(%%rdi)\n\t"
        "movdqu -48(%%rsi,%%rdx), %%xmm8\n\t"
        "movdqu %%xmm8, -48(%%rdi,%%rdx)\n\t"
        "movdqu 48(%%rsi), %%xmm8\n\t"
        "movdqu %%xmm8, 48(%%rdi)\n\t"
```

```
"movdqu -64(%%rsi,%%rdx), %%xmm8\n\t"
"movdqu %%xmm8, -64(%%rdi,%%rdx)\n\t"
"jbe 11b\n\t"
"leaq 64(%%rdi), %%rcx\n\t"
"addq %%rdi, %%rdx\n\t"
"andq $-64, %%rdx\n\t"
"andq $-64, %%rcx\n\t"
"movq %%rcx, %%rax\n\t"
"subq %%rdi, %%rax\n\t"
"addq %%rax, %%rsi\n\t"
"cmpq %%rdx, %%rcx\n\t"
"je 11b\n\t"
"movq %%rsi, %%r10\n\t"
"subq %%rcx, %%r10\n\t"
"leaq 16(%%r10), %%r9\n\t"
"leaq 32(%%r10), %%r8\n\t"
"leaq 48(%%r10), %%rax\n\t"
".p2align 4,,10\n\t"
".p2align 4\n\t"
"12:\n\t"
"movdqu (%%rcx,%%r10), %%xmm8\n\t"
"movdqa %%xmm8, (%%rcx)\n\t"
"movdqu (%%rcx,%%r9), %%xmm8\n\t"
"movdqa %%xmm8, 16(%%rcx)\n\t"
"movdqu (%%rcx,%%r8), %%xmm8\n\t"
"movdqa %%xmm8, 32(%%rcx)\n\t"
"movdqu (%%rcx,%%rax), %%xmm8\n\t"
"movdqa %%xmm8, 48(%%rcx)\n\t"
"addq $64, %%rcx\n\t"
"cmpq %%rcx, %%rdx\n\t"
"jne 12b\n\t"
"jmp 11b\n\t"
"8:\n\t"
"cmpq %%rsi, %%rdi\n\t"
"jae 13f\n\t"
"testq %%rdx, %%rdx\n\t"
".p2align 4,,5\n\t"
"je 11b\n\t"
"movq %%rdx, %%r9\n\t"
"leaq 16(%%rsi), %%rcx\n\t"
"leaq 16(%%rdi), %%r8\n\t"
"shrq $4, %%r9\n\t"
"movq %%r9, %%rax\n\t"
"salq $4, %%rax\n\t"
"cmpq %%rcx, %%rdi\n\t"
"setae %%cl\n\t"
"cmpq %%r8, %%rsi\n\t"
"setae %%r8b\n\t"
"orl %%r8d, %%ecx\n\t"
"cmpq $15, %%rdx\n\t"
"seta %%r8b\n\t"
"testb %%r8b, %%cl\n\t"
"je 19f\n\t"
"testq %%rax, %%rax\n\t"
"je 19f\n\t"
"xorl %%ecx, %%ecx\n\t"
```

```
"xorl %%r8d, %%r8d\n\t"
"14:\n\t"
"movdqu (%%rsi,%%rcx), %%xmm8\n\t"
"addq $1, %%r8\n\t"
"movdqu %%xmm8, (%%rdi,%%rcx)\n\t"
"addq $16, %%rcx\n\t"
"cmpq %%r8, %%r9\n\t"
"ja 14b\n\t"
"cmpq %%rax, %%rdx\n\t"
"je 11b\n\t"
"15:\n\t"
"movzbl (%%rsi,%%rax), %%ecx\n\t"
"movb %%cl, (%%rdi,%%rax)\n\t"
"addq $1, %%rax\n\t"
"cmpq %%rax, %%rdx\n\t"
"ja 15b\n\t"
"jmp 11b\n\t"
"9:\n\t"
"testb $24, %%dl\n\t"
"jne 16f\n\t"
"testb $4, %%dl\n\t"
".p2align 4,,5\n\t"
"jne 17f\n\t"
"testq %%rdx, %%rdx\n\t"
".p2align 4,,2\n\t"
"je 11b\n\t"
"movzbl (%%rsi), %%eax\n\t"
"testb $2, %%dl\n\t"
"movb %%al, (%%rdi)\n\t"
"je 11b\n\t"
"movzwl -2(%%rsi,%%rdx), %%eax\n\t"
"movw %%ax, -2(%%rdi,%%rdx)\n\t"
"jmp 11b\n\t"
"13:\n\t"
"leaq -1(%%rdx), %%rax\n\t"
".p2align 4,,10\n\t"
".p2align 4\n\t"
"18:\n\t"
"movzbl (%%rsi,%%rax), %%edx\n\t"
"movb %%dl, (%%rdi,%%rax)\n\t"
"subq $1, %%rax\n\t"
"jmp 18b\n\t"
"16:\n\t"
"movq (%%rsi), %%rax\n\t"
"movq %%rax, (%%rdi)\n\t"
"movq -8(%%rsi,%%rdx), %%rax\n\t"
"movq %%rax, -8(%%rdi,%%rdx)\n\t"
"jmp 11b\n\t"
"19:\n\t"
"xorl %%eax, %%eax\n\t"
"jmp 15b\n\t"
"17:\n\t"
"movl (%%rsi), %%eax\n\t"
"movl %%eax, (%%rdi)\n\t"
"movl -4(%%rsi,%%rdx), %%eax\n\t"
"movl %%eax, -4(%%rdi,%%rdx)\n\t"
```

```
        "jmp 11b\n\t"
        "20:\n\t"
        "movq %%rax, %[destination]\n\t"
        :[destination] "+r"(destination)
        /* + means that destination is both input
        and output operand  */
        :[source] "r"(source), [num] "r"(num)
        /* source and num are input operands */
        :"cc", "memory", "%rdi", "%rsi", "%rdx", "%rax", "%rcx", "%xmm8",
         "%r10", "%r9", "%r8", "%cl", "%ecx", "%dl", "%eax", "%edx"
        );

        return destination;
}
```

Appendix B

C Library Patches

B.1 Musl-Libc Patch

```
diff -r --unified '--exclude=.*' musl-1.1.20/arch/aarch64/syscall_arch.h "custom_musl-1.1_.20/arch/aarch64
    /syscall_arch.h"
--- musl-1.1.20/arch/aarch64/syscall_arch.h      2018-09-04 10:17:19.000000000 -0700
+++ "custom_musl-1.1_.20/arch/aarch64/syscall_arch.h"    2018-12-29 12:23:12.161871722 -0800
@@ -71,6 +71,6 @@
        __asm_syscall("r"(x8), "0"(x0), "r"(x1), "r"(x2), "r"(x3), "r"(x4), "r"(x5));
 }

-#define VDSO_USEFUL
-#define VDSO_CGT_SYM "__kernel_clock_gettime"
-#define VDSO_CGT_VER "LINUX_2.6.39"
+//#define VDSO_USEFUL
+//#define VDSO_CGT_SYM "__kernel_clock_gettime"
+//#define VDSO_CGT_VER "LINUX_2.6.39"
diff -r --unified '--exclude=.*' musl-1.1.20/arch/arm/syscall_arch.h "custom_musl-1.1_.20/arch/arm/
    syscall_arch.h"
--- musl-1.1.20/arch/arm/syscall_arch.h 2018-09-04 10:17:19.000000000 -0700
+++ "custom_musl-1.1_.20/arch/arm/syscall_arch.h"        2018-12-29 12:23:31.113858082 -0800
@@ -98,9 +98,9 @@
        __asm_syscall(R7_OPERAND, "0"(r0), "r"(r1), "r"(r2), "r"(r3), "r"(r4), "r"(r5));
 }

-#define VDSO_USEFUL
-#define VDSO_CGT_SYM "__vdso_clock_gettime"
-#define VDSO_CGT_VER "LINUX_2.6"
+//#define VDSO_USEFUL
+//#define VDSO_CGT_SYM "__vdso_clock_gettime"
+//#define VDSO_CGT_VER "LINUX_2.6"
```

```
 #define SYSCALL_FADVISE_6_ARG

diff -r --unified '--exclude=.*' musl-1.1.20/arch/i386/syscall_arch.h "custom_musl-1.1_.20/arch/i386/
    syscall_arch.h"
--- musl-1.1.20/arch/i386/syscall_arch.h        2018-09-04 10:17:19.000000000 -0700
+++ "custom_musl-1.1_.20/arch/i386/syscall_arch.h"      2018-12-29 12:23:57.657850095 -0800
@@ -52,8 +52,8 @@
        return __ret;
 }

-#define VDSO_USEFUL
-#define VDSO_CGT_SYM "__vdso_clock_gettime"
-#define VDSO_CGT_VER "LINUX_2.6"
+//#define VDSO_USEFUL
+//#define VDSO_CGT_SYM "__vdso_clock_gettime"
+//#define VDSO_CGT_VER "LINUX_2.6"

 #define SYSCALL_USE_SOCKETCALL
diff -r --unified '--exclude=.*' musl-1.1.20/arch/x86_64/syscall_arch.h "custom_musl-1.1_.20/arch/x86_64/
    syscall_arch.h"
--- musl-1.1.20/arch/x86_64/syscall_arch.h      2018-09-04 10:17:19.000000000 -0700
+++ "custom_musl-1.1_.20/arch/x86_64/syscall_arch.h"    2018-12-29 12:25:30.673915735 -0800
@@ -61,8 +61,8 @@
        return ret;
 }

-#define VDSO_USEFUL
-#define VDSO_CGT_SYM "__vdso_clock_gettime"
-#define VDSO_CGT_VER "LINUX_2.6"
-#define VDSO_GETCPU_SYM "__vdso_getcpu"
-#define VDSO_GETCPU_VER "LINUX_2.6"
+//#define VDSO_USEFUL
+//#define VDSO_CGT_SYM "__vdso_clock_gettime"
+//#define VDSO_CGT_VER "LINUX_2.6"
+//#define VDSO_GETCPU_SYM "__vdso_getcpu"
+//#define VDSO_GETCPU_VER "LINUX_2.6"
```

B.2 Glibc-2.25 Patch

```
diff -r --unified '--exclude=.*' glibc2.25-original/elf/dl-support.c glibc2.25-custom/elf/dl-support.c
--- glibc2.25-original/elf/dl-support.c 2019-03-05 16:38:03.946832807 -0800
+++ glibc2.25-custom/elf/dl-support.c   2019-03-05 17:25:35.688935557 -0800
@@ -258,11 +258,13 @@
 break;
 #ifdef NEED_DL_SYSINFO
 case AT_SYSINFO:
+    av->a_un.a_val = 0;
 GL(dl_sysinfo) = av->a_un.a_val;
 break;
 #endif
 #ifdef NEED_DL_SYSINFO_DSO
 case AT_SYSINFO_EHDR:
+    av->a_un.a_val = 0;
```

```
GL(dl_sysinfo_dso) = (void *) av->a_un.a_val;
break;
#endif
diff -r --unified '--exclude=.*' glibc2.25-original/elf/dl-sysdep.c glibc2.25-custom/elf/dl-sysdep.c
--- glibc2.25-original/elf/dl-sysdep.c  2019-03-05 16:38:03.946832807 -0800
+++ glibc2.25-custom/elf/dl-sysdep.c    2019-03-05 17:25:35.688935557 -0800
@@ -169,11 +169,13 @@
break;
#ifdef NEED_DL_SYSINFO
case AT_SYSINFO:
+    av->a_un.a_val = 0;
new_sysinfo = av->a_un.a_val;
break;
#endif
#ifdef NEED_DL_SYSINFO_DSO
case AT_SYSINFO_EHDR:
+    av->a_un.a_val = 0;
GLRO(dl_sysinfo_dso) = (void *) av->a_un.a_val;
break;
#endif
Only in glibc2.25-custom/elf: dl-sysdep.c.orig
diff -r --unified '--exclude=.*' glibc2.25-original/sysdeps/unix/sysv/linux/aarch64/sysdep.h glibc2.25-
    custom/sysdeps/unix/sysv/linux/aarch64/sysdep.h
--- glibc2.25-original/sysdeps/unix/sysv/linux/aarch64/sysdep.h 2019-03-05 16:38:04.730829263 -0800
+++ glibc2.25-custom/sysdeps/unix/sysv/linux/aarch64/sysdep.h   2019-03-05 17:25:35.688935557 -0800
@@ -153,9 +153,9 @@


/* List of system calls which are supported as vsyscalls.  */
-# define HAVE_CLOCK_GETRES_VSYSCALL     1
-# define HAVE_CLOCK_GETTIME_VSYSCALL    1
-# define HAVE_GETTIMEOFDAY_VSYSCALL     1
+// # define HAVE_CLOCK_GETRES_VSYSCALL 1
+// # define HAVE_CLOCK_GETTIME_VSYSCALL        1
+// # define HAVE_GETTIMEOFDAY_VSYSCALL 1

/* Define a macro which expands into the inline wrapper code for a system
call.  */
diff -r --unified '--exclude=.*' glibc2.25-original/sysdeps/unix/sysv/linux/arm/sysdep.h glibc2.25-custom/
    sysdeps/unix/sysv/linux/arm/sysdep.h
--- glibc2.25-original/sysdeps/unix/sysv/linux/arm/sysdep.h     2019-03-05 16:38:04.738829226 -0800
+++ glibc2.25-custom/sysdeps/unix/sysv/linux/arm/sysdep.h       2019-03-05 17:25:35.688935557 -0800
@@ -389,8 +389,8 @@
#define INTERNAL_SYSCALL_ERRNO(val, err)        (-(val))

/* List of system calls which are supported as vsyscalls.  */
-#define HAVE_CLOCK_GETTIME_VSYSCALL     1
-#define HAVE_GETTIMEOFDAY_VSYSCALL      1
+// #define HAVE_CLOCK_GETTIME_VSYSCALL 1
+// #define HAVE_GETTIMEOFDAY_VSYSCALL  1

#define LOAD_ARGS_0()
#define ASM_ARGS_0
diff -r --unified '--exclude=.*' glibc2.25-original/sysdeps/unix/sysv/linux/i386/sysdep.h glibc2.25-custom
    /sysdeps/unix/sysv/linux/i386/sysdep.h
--- glibc2.25-original/sysdeps/unix/sysv/linux/i386/sysdep.h    2019-03-05 16:38:04.754829153 -0800
```

```
+++ glibc2.25-custom/sysdeps/unix/sysv/linux/i386/sysdep.h      2019-03-05 17:25:35.688935557 -0800
@@ -308,8 +308,8 @@
__syscall_error (-(resultvar))

/* List of system calls which are supported as vsyscalls.  */
-# define HAVE_CLOCK_GETTIME_VSYSCALL     1
-# define HAVE_GETTIMEOFDAY_VSYSCALL      1
+// # define HAVE_CLOCK_GETTIME_VSYSCALL     1
+// # define HAVE_GETTIMEOFDAY_VSYSCALL      1

/* Define a macro which expands inline into the wrapper code for a system
call.  This use is for internal calls that do not need to handle errors
diff -r --unified '--exclude=.*' glibc2.25-original/sysdeps/unix/sysv/linux/x86_64/sysdep.h glibc2.25-
    custom/sysdeps/unix/sysv/linux/x86_64/sysdep.h
--- glibc2.25-original/sysdeps/unix/sysv/linux/x86_64/sysdep.h  2019-03-05 16:38:04.814828883 -0800
+++ glibc2.25-custom/sysdeps/unix/sysv/linux/x86_64/sysdep.h    2019-03-05 17:25:35.688935557 -0800
@@ -257,9 +257,9 @@
# define INTERNAL_SYSCALL_ERRNO(val, err)       (-(val))

/* List of system calls which are supported as vsyscalls.  */
-# define HAVE_CLOCK_GETTIME_VSYSCALL     1
-# define HAVE_GETTIMEOFDAY_VSYSCALL      1
-# define HAVE_GETCPU_VSYSCALL            1
+// # define HAVE_CLOCK_GETTIME_VSYSCALL     1
+// # define HAVE_GETTIMEOFDAY_VSYSCALL      1
+// # define HAVE_GETCPU_VSYSCALL                1

# define LOAD_ARGS_0()
# define LOAD_REGS_0
```

Acronyms

NVX N-Variant Execution

TCB Trusted Computing Base

CFI Control-Flow Integrity

DOP Data-Oriented Programming

PIROP Position-Oriented Programming

PISC Platform-Independent State Canonicalization

RB Replication Buffer

CB Communication Buffer

IP-MON In-Process Monitor

CP-MON Cross-Process Monitor

K-IP Kernel Space In-Process

U-CP User Space Cross-Process

U-IP User Space In-Process

H-IP Hypervisor-Space In-Process

DSL Domain-Specific Language

IPC Inter-Process Communication

ISA Instruction Set Architecture

ABI Application Binary Interface

RDMA Remote Direct Memory Access

DEP Data Execution Prevention

ASLR Address Space Randomization

KTCP Kernel Space TCP

UTCP User Space TCP

ACC Asynchronous Cross-Checking

PFA Permissive Filesystem Access

DIP-MON Distributed In-Process Monitor

DCP-MON Distributed Cross-Process Monitor

CET Control-Flow Enforcement Technology

MPK Memory Protection Keys

RSM Replicated State Machines

BFT Byzantine Fault Tolerance

SCC Selective Cross-Checking

ESR Extensive Selective Replication

ASP Address Space Partioning

ROP Return-oriented programming

"And once the storm is over, you won't remember how you made it through, how you managed to survive. You won't even be sure, whether the storm is really over. But one thing is certain. When you come out of the storm, you won't be the same person who walked in. Thats what this storm is all about."

–Haruki Murakami