Geavanceerde technieken voor de uitvoering van meerdere varianten

Advanced Techniques for Multi-Variant Execution

Stijn Volckaert

Promotoren: prof. dr. ir. B. De Sutter, prof. dr. ir. K. De Bosschere
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. R. Van de Walle
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2015 - 2016

UNIVERSITEIT GENT

ISBN 978-90-8578-858-4
NUR 980
Wettelijk depot: D/2015/10.500/102

Dankwoord

Eerst en vooral wil ik graag Bjorn en Koen bedanken om mij de kans te geven om in hun groep te doctoreren. Bjorn, bedankt ook voor al je advies, geduld, aanmoedigingen en alle dingen die ik van jou heb kunnen leren. Koen, jou wil ik bedanken voor de vele rondjes feedback op mijn papers, verslagen en projectvoorstellen.

Next, I would like to thank the other members of my examination board: Prof. Gert De Cooman, Dr. Bart Coppens, Prof. Filip De Turck, Prof. Frank Piessens, Prof. Jack Davidson, and Prof. Michael Franz. I especially thank Jack for travelling all the way to Ghent to attend my internal defense.

Uiteraard wil ik ook het Agentschap voor Innovatie door Wetenschap en Technologie (IWT) bedanken voor hun financiële steun.

Daarnaast wil ik ook mijn vele (ex-)collega's van het System Software Lab bedanken om allerhande redenen. Hierbij denk ik onder andere aan Bart voor al zijn constructieve input en advies, Christophe en Jeroen voor de vele "naschoolse" bowlingsessies, Ronald, Bert en Tim voor de hulp bij de Werkgroep "Ethical" Hacking, Panagiotis om ons uit te nodigen op zijn onvergetelijk trouwfeest, Niels voor zijn steeds spectaculaire uiteenzettingen op de intussen legendarische team meetings, Jonas voor de hulp met Diablo en om ons allen inzicht te verschaffen in copyright, intellectuele eigendom, DRM en andere onderwerpen des duivels, Jens om vrijwillig te poseren voor de eerste jaarlijkse Open System Software Lab (OpenSSL) fotoshoot, Michiel, Ronny en Marnix voor de fantastische technische ondersteuning, en verder ook Hadi, Wim, Henri, en Sander om samen met de andere collega's een aangename werksfeer te creëren. Alle bovengenoemde personen wil ik ook bedanken om mijn eindeloze rants over Linux, Glibc, Pthreads, Java, en andere technologisch inferieure producten te tolereren.

Ik dank verder ook Prof. Lieven Eeckhout om hardware ter beschikking te stellen voor mijn onderzoek. Ook Wim Heirman en Sam Van den Steen ben ik zeer dankbaar om hierbij de nodige technische ondersteuning te bieden.

Next, I would like to thank Prof. Michael Franz for inviting me to visit his research group at the University of California, Irvine (UCI). I thoroughly enjoyed my stay at UCI and it was a great pleasure to work alongside Per, Stefan, Stephen, Andrei, Mark, Julian, Gülfem, Codruţ, Brian, Mohaned, and Wei. During my stay at UCI, I also had the privilege to collaborate with Felix Schuster, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. I thank you all for the excellent work on the Readactor++ paper.

Verder wil ik ook mijn vrouw Evelien, mijn ouders Ferdi en Kathleen, mijn broers Domien en Maarten, en mijn zus Laura, bedanken. Jullie hebben onrechtstreeks misschien wel de grootste bijdrage geleverd aan mijn doctoraat. Jullie hebben me de laatste jaren door dik en dun gesteund. Mama en Papa, jullie hebben me tijdens mijn studies steeds weer aangespoord om toch maar te studeren voor mijn examens. Zonder jullie had mijn leven er nu wellicht heel anders uitgezien. Domien en Maarten, onze minivoetbalwedstrijden en gezamenlijke uitstapjes naar de matchen van de Gantoise waren een broodnodige en zeer geapprecieërde afwisseling van het werk. Laura, jouw bezoekjes waren altijd weer hoogtepunten. Dankzij jou ben ik nu ook een specialist in *Rayman* en weet ik alles over neusfluiten. Evelien, bedankt om er altijd voor mij te zijn en om binnenkort samen met mij in een nieuw avontuur te stappen.

Tenslotte dank ik ook mijn (ex-)ploegmaats bij *The Black Adders* - Jonas, Tim, Kilian, Roeland, Christof, Cedric, Ben, Jeroen, en Simon, mijn niet-voetballende vrienden - Jer, Leen, Ine, Renaat, Dave, Viktor, en Hannes - en iedereen die ik onbewust over het hoofd gezien heb in dit dankwoord.

Stijn Volckaert
Gent, 23 Oktober 2015

Examencommissie

Leescommissie

Samenvatting

Multi-Variante Uitvoeringsomgevingen (MVUOs) zijn een veelbelovende techniek om software te beschermen tegen op geheugencorruptie gebaseerde aanvallen. MVUOs voeren op een transparente manier meerdere, gediversifieerde varianten (ook wel replicae genoemd) van hetzelfde programma uit, en zorgen ervoor dat deze replicae dezelfde invoer krijgen. Door de replicae in *lockstep* uit te voeren en te monitoren op het niveau van systeemoproepen, en door diversificatietechnieken te gebruiken die ervoor zorgen dat aanvallers geen meerdere replicae tegelijk kunnen compromitteren, zorgen MVUOs ervoor dat aanvallen gestopt worden voor ze schade kunnen toebrengen.

Verschillende problemen zorgen er echter voor dat MVUOs nog niet algemeen gebruikt kunnen worden. In dit proefschrift stel ik verschillende technieken voor die bestaande MVUOs kunnen verbeteren en zo deze problemen kunnen verlichten. Ik pas deze technieken toe in mijn eigen MVUO prototype, GHUMVEE.

Inconsistenties en Vals-Positieve Detecties

Replicae die binnen een MVUO uitgevoerd worden moeten consistente invoer krijgen opdat ze zich identiek zouden gedragen. Replicae die verschillende invoer krijgen kunnen namelijk andere sequenties van systeemoproepen uitvoeren, en daardoor dus vals-positieve detecties veroorzaken in GHUMVEE. Oudere MVUOs gaan ervan uit dat alle programmainvoer ofwel aan de hand van systeemoproepen wordt gelezen, of dat de invoer aan banden gelegd kan worden aan de systeeminterface, bijvoorbeeld door het gebruik van gedeeld geheugen te verbieden. Wij betwisten deze aanname.

Ten eerste zijn er verschillende programma's die *impliciete programmainvoer* zoals numerieke pointerwaarden lezen. Deze numerieke pointerwaarden verschillen hoogstwaarschijnlijk wanneer meerdere replicae naast elkaar worden uitgevoerd, omdat de replicae ge-

diversifieerd zijn. We hebben verschillende situaties geïdentificeerd waarin programmeurs impliciete programmainvoer gebruiken. We stellen voor om *replicatiebemiddelaars voor impliciete invoer* te gebruiken om ervoor te zorgen dat alle replicae dezelfde impliciete invoer lezen. Deze bemiddelaars leggen de impliciete invoer vast in één replica, de meester, en sturen die invoer door naar de andere replicae, de slaven.

Ten tweede stellen x86 processors tijdsinformatie ter beschikking van uitvoerende programma's. Deze informatie kan uitgelezen worden met niet-bevoorrechte machineinstructies. Deze instructies kunnen buiten het gezichtsveld van de MVUO uitgevoerd worden. We stellen daarom voor om gebruik te maken van de *TimeStamp Disable* capabiliteit van de processor om zo alle uitvoeringen van de deze instructies te onderscheppen en te emuleren in de monitor van de MVUO.

Ten derde maken moderne versies van Linux gebruik van virtuele systeemoproepen. Virtuele systeemoproepen worden niet gemeld aan de monitor van de MVUO. Deze virtuele oproepen moeten wel afkomstig zijn uit het Virtual Dynamic Shared Object (VDSO). We stellen daarom voor om deze VDSO te verbergen voor de replicae, waardoor ze automatisch terugvallen op niet-virtuele systeemoproepen, die wel gemeld worden aan de monitor.

Ten laatste geven we een uitgebreider overzicht van andere bronnen van programmainvoer dan er in de literatuur beschikbaar is. We stellen technieken voor die deze andere bronnen ofwel elimineren, ofwel de MVUO toelaten om ze te tolereren.

Afleveren van Asynchrone Signalen

MVUOs die hun replicae in *lockstep* uitvoeren moeten voorzichtig omspringen met asynchrone signalen. Asynchrone signalen die naar de replicae gestuurd worden, moeten afgeleverd worden op hetzelfde punt hun uitvoering. Gebeurt dit niet, dan kunnen er inconsistenties optreden en. De complexiteit van de ptrace interface, het feit dat signalen voor allerhande uiteenlopende doeleinden gebruikt worden, en de vele scenarios waarin signalen afgeleverd kunnen worden, zorgen er echter voor dat het correct afhandelen van asynchrone signalen een grote uitdaging is bij het implementeren van MVUOs. Het trieste gevolg is dat bijna geen enkele van de bestaande MVUOs het afleveren van signalen volledig ondersteunen. Wij bieden een overzicht van de interactie tussen processen die signalen ontvangen en hun debuggers, en beschrijven alle scenarios waarin de MVUO nauwgezet moet optreden. GHUMVEE is, voor zover wij weten, de enige op veiligheid gerichte MVUO die volledige ondersteuning biedt voor asynchrone signalen.

Multi-Variante Uitvoering van Parallelle Programma's

Vele parallelle programma's voeren van nature niet-deterministisch uit en lijken zich daarom bij elke uitvoering anders te gedragen als ze van aan de systeeminterface geobserveerd worden. Dit nietdeterminisme wordt veroorzaakt door de werkverdeling. In parallelle programma's die vrij kunnen uitvoeren, ligt de volgorde waarin geheugentoegangen naar gedeeld geheugen uitgevoerd worden niet vast. Hierdoor kan de volgorde waarin de draden van het programma de effecten van die geheugentoegangen zien bij elke uitvoering verschillen.

Deterministic MultiThreading (DMT) en Record+Replay (R+R) systemen zijn twee bestaande oplossingen die niet-deterministisch gedrag ofwel kunnen elimineren ofwel kunnen repliceren in meerdere uitvoeringen van eenzelfde programma. DMT systemen leggen een deterministische volgorde op aan alle geheugentoegangen naar gedeeld geheugen. Deze deterministische volgorde baseren ze op een planning die voor elke programmainvoer vastgelegd wordt. Bij bepaalde DMT systemen kan deze planning echter niet vastgelegd worden voor parallelle programma's waarin sommige draden oneindige lussen uitvoeren of systeemoproepen gebruiken die voor onbepaalde duur blokkeren. Bij andere DMT systemen is de vastgelegde planning nauw verbonden aan bepaalde eigenschappen van het programma, die naar alle waarschijnlijkheid veranderen wanneer diversificatietechnieken toegepast worden.

R+R systemen lijden niet aan voorgenoemde beperkingen maar ze moeten wel aangepast worden om binnen een MVUO bruikbaar te zijn. Concreet moeten R+R systemen ongevoelig gemaakt worden voor de precieze geheugenlay-out van het programma waarin ze gebruikt worden. Verder moeten de R+R systemen een neutraal effect hebben op het van aan de systeeminterface observeerbare gedrag van het programma. Tenslotte moeten bestaande R+R systemen ook uitgebreid worden om ad hoc-synchronizatie te ondersteunen.

Wij stellen vier op R+R-gebaseerde synchronizatiebemiddelaars voor die voldoen aan deze vereisten. Onze synchronizatiebemiddelaars leggen de volgorde vast waarin één replica, de meester, zijn synchronizatieoperaties uitvoert. Een equivalente volgorde wordt dan opgelegd aan de andere replicae, de slaven. We stellen verder praktisch toepasbare strategieën voor om de synchronizatiebemiddelaars in te bouwen in programma's en in programmabibliotheken die van ad hoc-synchronizatie gebruik maken.

Onze bemiddelaars maken van GHUMVEE de eerste MVUO die, met een geringe inspanning, arbitraire parallelle programma's ondersteunt. Onze *wall-of-clocks* synchronizatiebemiddelaar is efficiënt en schaalbaar. Bij het uitvoeren van de PARSEC evaluatiesuite voor parallelle programma's, met vier *worker* draden en twee replicae, vertraagt deze *wall-of-clocks* bemiddelaar de uitvoering met een factor 1,32. Dit resultaat is vergelijkbaar met de vertragingen die door de auteurs van andere MVUOs gerapporteerd werden voor enkeldradige evaluatieprogramma's.

Disjoint Code Layouts

Dezelfde software die wij wensen te beschermen met onze MVUO, wordt regelmatig door hackers aangevallen met aanvallen door codehergebruik (*code reuse attacks*). Bij zulke aanvallen wordt een programma omgeleid van zijn bedoelde controleverloop naar een set van bekende locaties. Het doel van aanvallen door codehergebruik is om kwaadaardige acties uit te voeren binnen de context van het aangevallen programma.

Address Space Partitioning (ASP) is een bestaande diversificatietechniek die de meeste aanvallen door codehergebruik onmogelijk maakt. ASP deelt de virtuele adresruimte van de replicae op in n partities, waarbij n het aantal gelijktijdig uitgevoerde replicae is. Elke replica wordt begrensd zodat deze enkel van zijn eigen partitie kan gebruik maken.

ASP vereist echter een verschillend programmabestand voor elke replica en verlaagt de hoeveelheid beschikbaar virtueel geheugen voor elke replica met een factor n. Wij vinden daarom dat ASP niet praktisch bruikbaar is en stellen *Disjoint Code Layouts* (DCL) voor als alternatief. DCL zorgt ervoor dat de uitvoerbare coderegio's van alle replicae volledig disjunct zijn. Concreet wil dit zeggen dat een gegeven virtueel geheugenadres in ten hoogste één replica naar een geldige coderegio wijst. DCL grijpt hiertoe in in het laadproces voor alle uitvoerbare bestanden die in de virtuele adresruimten van de replicae geladen worden. DCL gebruikt één Position-Independent Executable (PIE) programmabestand voor alle replicae en heeft een minimale impact op de hoeveelheid beschikbaar virtueel geheugen.

DCL biedt dezelfde veiligheidsgaranties als ASP en is zeer efficiënt. Wij rapporteren vertragingen van slechts 6.37% voor de SPEC CPU

2006 evaluatiesuite die bovenop een 64-bit Linux 3.13 besturingssysteem wordt uitgevoerd.

Gerelaxeerd Monitoren

Op veiligheid gerichte MVUOs, zoals GHUMVEE, voeren de replicae en de monitor in afzonderlijke processen uit. Dit ontwerp komt de veiligheid duidelijk ten goede omdat gecompromitteerde replicae de monitor niet rechstreeks kunnen aanvallen. Het ontwerp is echter niet bevorderlijk voor de prestaties van de replicae omdat het veel vertraging veroorzaakt op de momenten dat de replicae en de monitor interageren.

Wij stellen daarom een nieuw concept voor, gerelaxeerd monitoren, en passen het toe in een nieuw ontwerp met een gesplitste monitor. Dit nieuw ontwerp, dat we ReMon noemen, gebruikt GHUMVEE als een traditionele monitor die als een afzonderlijk proces uitgevoerd wordt en die enkel de veiligheidsgevoelige systeemoproepen monitort. De overgebleven systeemoproepen handelen we af in een nieuwe monitor, IP-MON, die ingebouwd kan worden in de replicae.

We stellen ook verschillende beleidsplannen voor voor deze gerelaxeerde monitoring en evalueren deze plannen. De beleidsplannen bepalen de verantwoordelijkheden van elke component binnen ReMon. We concluderen dat ReMon significant sneller is dan de bestaande op veiligheid gerichte MVUOs, en dat ReMon tegelijk vergelijkbare veiligheidsgaranties biedt als deze bestaande MVUOs.

Summary

Multi-Variant Execution Environments (MVEEs) are a promising technique to protect software against memory corruption attacks. They transparently execute multiple, diversified variants (often referred to as replicae) of the software receiving the same inputs. By enforcing and monitoring the lock-step execution of the replicae's system calls, and by deploying diversity techniques that prevent an attacker from simultaneously compromising multiple replicae, MVEEs can block attacks before they succeed.

However, several problems stand in the way of widespread deployment of MVEEs in production environments. In this dissertation, I present several techniques that enhance existing MVEEs to alleviate these problems. I apply these techniques in my own proof-of-concept MVEE, GHUMVEE.

Inconsistencies and False-Positive Detections

Program replicae that run inside an MVEE must be fed consistent inputs in order to guarantee that they will behave the same. Replicae that receive different inputs might invoke different sequences of system calls and trigger false-positive detections in GHUMVEE as a consequence. Older MVEEs build on the assumption that all program inputs either originates from the system call interface, or can be stopped at the system call interface by, e.g., disallowing the use of shared memory. We argue that this assumption is false.

First, several programs rely on *implicit program inputs*, such as numerical pointer values. These numerical pointer values likely differ between replicae as a result of diversification. We identify several programming idioms in which implicit inputs are used and propose to deploy *implicit input replication agents* in order to force all replicae to use the same implicit inputs. These agents record the implicit inputs in one replica, the master, and forward them to the other replicae, the slaves.

Second, x86 processors expose (mutable) timing information through unprivileged machine instructions. These instructions can be executed without being supervised by the MVEE's monitor. We propose to leverage the processor's *TimeStamp Disable* capability to intercept all invocations of such instructions and to emulate them inside the MVEE's monitor.

Third, modern versions of Linux use virtual system calls, which are not reported to the MVEE's monitor. These virtual system calls must originate from the Virtual Dynamic Shared Object (VDSO), which is loaded into every running program's address space by the kernel. We propose to hide this VDSO from the replicae in the MVEE, thereby forcing them to fall back to regular system calls.

Finally, we provide a more extensive overview of other sources of program input than is available in the literature, and propose techniques that either eliminate these sources or that allow the MVEE to tolerate them.

Asynchronous Signal Delivery

Security-oriented MVEEs that run the replicae in lock-step must handle asynchronous signals carefully. Asynchronous signals sent to the replicae must be delivered when the replicae are at the same point in their execution, so as to avoid violations of the lock-step mechanism. The many intricacies of the ptrace API, the one-size-fits-all nature of signals in general, and the myriad of scenarios in which signals may be delivered make correct handling of asynchronous signals one of the most challenging problems when implementing an MVEE. The sad consequence is that almost no existing MVEEs fully support signal handling. We provide an extensive overview of the interaction between processes that receive signals and their debuggers, and describe the scenarios that an MVEE has to handle carefully. GHUMVEE is, to the best of our knowledge, the only security-oriented MVEE to fully support asynchronous signal delivery.

Multi-Variant Execution of Parallel Programs

Many parallel programs are non-deterministic by nature and will appear to behave differently from run to run when observed from the system call interface, even if they are not diversified. Scheduling is the root cause of this non-determinism. If parallel programs are allowed to run freely, the order in which they execute instructions that participate in inter-thread communication, or the order in which the effects of these instructions become visible to other threads will change from run to run.

Deterministic MultiThreading (DMT) and Record+Replay (R+R) systems are existing solutions to eliminate or replicate non-deterministic behavior. DMT systems impose a deterministic order on inter-thread communication instructions by establishing a fixed schedule for each given program input. Some DMT systems cannot establish such a schedule for programs with threads that perform unbounded computations or indefinitely blocking system calls. Other DMT systems establish a schedule that is tightly bound to program properties that are likely to change as a result of diversification.

R+R systems do not suffer from the same issues but need to be adapted before they can be used in the context of an MVEE. Specifically, R+R agents need to be address-agnostic, neutral with respect to system call behavior and activities that require intervention from our replication agents, and support ad hoc synchronization.

We present four R+R-based synchronization replication agents that fit within these constraints. Our synchronization replication agents capture the order in which synchronization operations are executed in the master replica, and force an equivalent order in the slave replicae. We also recommended practical strategies to embed our agents into programs and libraries that use ad hoc synchronization.

Our replication agents make GHUMVEE the first MVEE to support arbitrary multi-threaded replicae with limited effort. Our wall-of-clocks replication agents are efficient and scalable. When running the PARSEC benchmark suite with four worker threads and two replicae, the wall-of-clocks agents achieve slowdowns of just 1.32x, which is comparable to the slowdowns reported by authors of older MVEEs for single-threaded benchmarks.

Disjoint Code Layouts

The same software we wish to protect using our MVEE, is frequently targeted by hackers with code reuse attacks. Such attacks diverge the intended control flow of the target to a (set of) known location(s) so as to perform malicious actions chosen by the attacker in the context of the target program.

Address Space Partitioning (ASP) is an existing diversification technique that prevents attackers from successfully launching most code reuse attacks against replicae in an MVEE. ASP splits the replicae's address spaces into n partitions, with n the number of concurrently executing replicae, and confines each replica to its own partition.

ASP requires different program binaries for each replica and reduces the available amount of virtual memory in the replicae by a factor n. We therefore argue that ASP is impractical and propose Disjoint Code Layouts (DCL) as an alternative. DCL ensures that the replicae's executable code regions are fully disjoint, i.e., that any given virtual address cannot point to a valid executable memory page in more than replica. DCL achieves this by transparently manipulating the loading process for all executable files that are loaded into the replicae's address spaces. DCL uses a single Position-Independent Executable (PIE) program binary for all replicae and has a minimal impact on the available amount of virtual memory.

DCL achieves the same protection strength as partitioning, and is also efficient. We report performance overheads as low as 6.37% for the SPEC CPU 2006 benchmark suite running on top of a 64-bit Linux 3.13 OS.

Monitoring Relaxation

Security-oriented MVEEs, including GHUMVEE, run the replicae and the monitoring component as separate processes. This approach has obvious security benefits as compromised replicae cannot attack the monitor directly. The replicae's interaction with a monitor in a separate process does however significantly degrade their performance.

We therefore propose the concept of monitoring relaxation and apply it in a split-monitor design, called ReMon. In ReMon, we use GHUMVEE as a traditional monitor that runs as a separate process and that monitors security-sensitive system calls, and we use a new monitor called IP-MON as a component that can be embedded into the replicae and that can monitor innocuous system calls and replicate their results.

We propose and evaluate several monitoring relaxation policies that define the responsibilities of each component within ReMon. We conclude that ReMon vastly outperforms existing security-oriented MVEEs, while offering comparable security guarantees.

Contents

List of Tables

List of Figures

List of Abbreviations

I/O	Input/Output
MVEE	Multi-Variant Execution Environment
GHUMVEE	GHent University Multi-Variant Execution Environment
OS	Operating System
RVP	Rendez-Vous Point
IPC	Inter-Process Communication
RDTSC	ReaD TimeStamp Counter
VDSO	Virtual Dynamic Shared Object
ASB	Address-Sensitive Behavior
IP	In-Process
CP	Cross-Process
DCL	Disjoint Code Layouts
ASP	Address Space Partitioning
DMT	Deterministic MultiThreading
R+R	Record+Replay

Chapter 1

Introduction

Software vulnerabilities have become a major threat to our society. In the past few years, we have seen several high profile attacks that have crippled critical infrastructure or that have caused data breaches. This year alone, millions of people have seen their personal, medical and financial information leak to the Internet. Although we have successfully eliminated certain classes of vulnerabilities, the outlook for the next few years is relatively bleak and we should expect more incidents in the foreseeable future. The root cause for this problem is that a lot of programmers still use unsafe languages such as C and C++ to write their software. While many of the alternatives for C and C++ offer type-safety, high productivity and reasonable performance, none of them have been able to make a big impact in low-level software. This is not entirely surprising. C and C++ offer a combination of explicit memory management, direct access to hardware, and high performance that has not been paralleled by any other language.

1.1 Memory Vulnerabilities, Attacks and Defenses

The vulnerabilities that have plagued unsafe languages for decades are well understood in the hacker community and often allow attackers to take over control of a program. We distinguish between two classes of vulnerabilities. Spatial memory vulnerabilities such as buffer overflows result from poor (or complete lack of) bounds checking when reading from or writing into a fixed-size object [126]. Temporal memory vulnerabilities such as use-after-free and double-free conditions result from using (e.g., dereferencing) invalid pointers.

Aleph One demonstrated how to exploit spatial memory vulnerabilities as early as 1996 [4]. His technique, known as "stack smashing", works by injecting malicious code into a running program and by diverging the program's control flow to the injected code. The security community countered such attacks by forcing data-only memory regions, such as the stack, to be non-executable. This anti-code injection technique is now known as W$\oplus$X or Data Execution Prevention (DEP) [103].

In 1997, shortly after the initial version of W$\oplus$X was introduced, Solar Designer proposed to circumvent it by reusing code from the application itself to mount attacks. Solar Designer's "return-into-libc" (RILC) technique works by overwriting the topmost stack frame of the target application, including the return address, thus hijacking the application's control flow when it executes its next return instruction [125]. RILC attacks overwrite the return address with the address of a known function and write malicious parameters for this function onto the stack.

The security community responded by deploying Address Space Layout Randomization (ASLR) [102] and Stack Canaries [34]. ASLR randomizes the base addresses of a program's stack, heap and code segments [102]. It therefore reduces the chance of a successful RILC attack by forcing the attacker to guess the location of the function(s) he wishes to invoke. However, ASLR has only been mildly successful. In 2004, Shacham pointed out that ASLR lacks entropy on 32-bit systems and can therefore be bypassed in a matter of minutes [122]. Furthermore, compilers on the GNU/Linux platform historically generated program binaries that had to be loaded at a fixed address and were therefore not subject to ASLR. All of the industry-standard compilers can now generate Position Independent Executables (PIE) that are subject to ASLR, but none generate them by default. Stack Canaries are an ad hoc defense mechanism against stack smashing. They have however been bypassed as early as 2002 [109], and can be leaked through information leakage attacks [19].

In 2001, Nergal presented an advanced version of the RILC attack [95]. Nergal's attack was able to invoke a chain of RILC calls, whereas the original technique was limited to one call. Nergal also proposed several methods to bypass ASLR.

Control-Flow Integrity (CFI) was introduced to combat code reuse attacks in 2005 [1]. CFI instruments all indirect branch instructions as well as function return instructions in a program and checks whether their targets adhere to the intended control flow graph of the program. While a promising technique, CFI as well as its recent variations [90,137,143,144] suffer from two fundamental issues. First, the precise control flow graph for a program typically cannot be obtained without access to the program sources. Second, there is a high performance cost associated with enforcing strict control flow integrity [47].

In 2007, Shacham introduced Return-Oriented Programming (ROP), a much more powerful code reuse attack [121]. ROP attacks overwrite the target application's stack with a contiguous list of addresses of small code fragments (often referred to as gadgets). These gadgets must end with a return instruction so that each gadget automatically invokes the next gadget in the list. The address of the first gadget in the list must be written at the location of the return address in the program's topmost stack frame such that when the program executes its next return instruction, it automatically invokes the entire list of gadgets. Shacham showed that in any reasonably sized code base, enough gadgets could be found for the attacker to write Turing-complete programs consisting only of gadgets, thus allowing the attacker to perform arbitrary computations. ROP has inspired recent attacks such as Jump-Oriented Programming (JOP) [20] and ROP without returns [28]. It has also inspired the TC-RILC proposal, a generalized and Turing-complete variant of Nergal's advanced RILC attack [128].

Recent research has shown that ROP attacks can bypass CFI-based defenses by cleverly choosing the gadgets [26,38,47,116].

As an alternative to the above attack techniques, Schuster et al. recently proposed a whole-function reuse attack called Counterfeit Object-Oriented Programming (COOP) [115]. COOP can exploit spatial as well as temporal memory vulnerabilities. COOP overwrites C++ objects, including their virtual function tables, and then invokes functions in those tables using an invalid pointer to the overwritten object.

1.2 Multi-Variant Execution

Despite the successful deployment of defensive countermeasures such as ASLR, W$\oplus$X and Stack Canaries, hackers are still managing to set up attacks on widely used software on a daily basis. Researchers and industry professionals alike acknowledge this problem and are actively looking into new ways to secure software without forcing a move to-

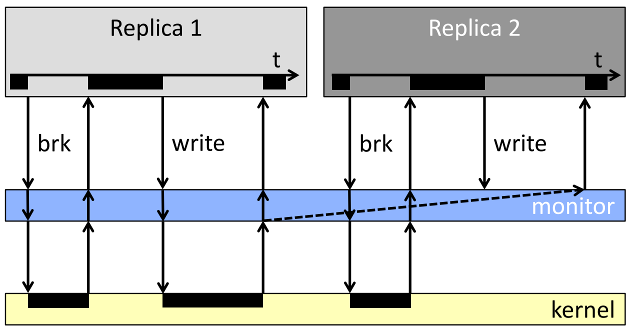


Figure 1.1: Basic operation of a MVEE

wards safer languages. So far though, their efforts have only been mildly successful.

Defensive countermeasures that enforce complete spatial [93] or temporal [94] memory safety incur excessive performance overhead. Approaches that incur less overhead have been bypassed or are limited in scope or impractical to deploy.

Redundant Execution is a class of techniques that aim to alleviate this problem. These techniques can serve many purposes. They have been instrumental in the field of Software-based Fault Tolerance [11,21,22,135] and have recently been used for live patch testing and debugging [52,53]. When combined with software diversity, however, they can be a powerful mechanism to protect software against memory-based exploits. This combination of redundant execution and software diversity is commonly referred to as Multi-Variant Execution. In a Multi-Variant Execution Environment (MVEE), several instances of the same program run side by side. Since these instances are not identical, they are often referred to as either replicae or variants. A monitor, the MVEE's main component, ensures that all replicae are fed the same input. It then continuously monitors the replicae' behavior. Security-oriented MVEEs typically perform these tasks by running the replicae in lock-step at system call granularity, as illustrated in Figure 1.1, but other options are possible.

MVEEs heavily rely on the principle of asymmetrical attacks. By transforming the replicae in a specific way, either at compile time or at run time, an MVEE can ensure that even if an attack can successfully compromise one of the replicae, it will not have the desired effect on the other replicae. When attacked, these other replicae will either crash or will not be affected by the attack at all. In both cases, the MVEE's monitor will be able to observe differences in the replica's future system call behavior.

The security guarantees provided by an MVEE build on three properties: (i) isolation of the monitor from the replicae; (ii) monitored lock-step system call execution; and (iii) diversification of the replicae. The isolation of the monitor by means of hardware-enforced boundaries is achieved by implementing it in kernel-space [35] or in a separate user-space process [27,52,85,114,133]. All system calls invoked in the replicae are monitored, executed in lock-step, and only allowed to execute when all replicae invoke the same system calls with consistent inputs. This allows the monitor to detect when a single replica is compromised and to halt its execution. However, this also implies that all replicae only progress at the speed of the slowest one. Furthermore, it implies that the monitor cannot look ahead at future system calls when deciding if a specific system call invocation should be allowed. This places strict constraints on the system calls in the replicae. Foremost, they need to occur in the same order in all replicae.

Several diversification techniques have been used in existing MVEEs to generate the replicae. Instruction Set Tagging was proposed as an anti-code injection technique by Cox et al. [35]. Salamat used System Call Number Randomization to prevent code injection [32]. Salamat et al. also proposed Reverse Stack Growth as a spatial memory safety technique [112]. Cox et al., as well as Cavallaro, independently proposed to confine each replica to a fixed partition in the virtual memory space [27, 35]. In Chapter 5 finally, I propose a practical and enhanced version of this partitioning technique.

MVEEs have several advantages over existing defensive techniques. First and foremost, they do not embed costly security checks in the programs they protect. Instead, they only apply transformations that ensure that the replicae will behave differently when being attacked. These transformations typically come with little or no cost in terms of run-time performance.

Second, several of the diversification techniques that have been used in existing MVEEs are precise and transparent. These diversification techniques can completely eliminate certain attack vectors and can be applied even if the source code of the protected program is not available. Techniques such as CFI by contrast typically lose precision due to the absence of source code information [1,90,143,144].

Third, MVEEs are efficient. The authors of Orchestra, which was the state-of-the-art MVEE for several years, established that an MVEE could run two replicae of a single-threaded program while incurring a slowdown of less than 20%. At the time, this slowdown was comparable to other state-of-the-art defensive techniques such as CFI. This situation has not changed much since then. Though MVEEs can certainly boast good results in terms of performance impact, there is a hidden cost. The system-wide CPU utilization, memory consumption and power consumption all scale linearly with the number of replicae that are simultaneously running. However, as the industry is evolving towards machines with more and more CPU cores and as memory becomes bigger and cheaper, we believe that this hidden cost is acceptable for applications that require strong security guarantees. Furthermore, we assume that the true cost of Multi-Variant Execution is minimal for server applications because, typically, the kernel-space portion of the replicae is executed only once in an MVEE and server applications spend a significant fraction of their time in kernel space. We also assume that server applications typically run on over-provisioned server blades. We leave the confirmation of these assumptions as future work.

Several problems stand in the way of the widespread deployment of MVEEs, however. None of the many MVEEs that have been proposed in the past decade could run replicae that execute non-deterministically ,i.e., replicae that do not always execute the same sequence of system calls, when given a certain input. Further, existing MVEEs offer limited or no protection against modern code reuse attacks such as ROP [24, 121] or COOP [115]. Finally, all of the existing MVEEs impose too much run-time overhead to be useful in a production environment.

In this dissertation, I present the GHent University Multi-Variant Execution Engine (GHUMVEE). GHUMVEE is a security-oriented MVEE for the GNU/Linux platform. It supports the 32-bit and 64-bit x86 architectures. GHUMVEE tackles all of above problems and could be a major step towards widespread deployment of MVEEs.

1.3 Contributions and Structure of the Dissertation

In this dissertation I contribute the following:

In Chapter 2, I describe the high-level design principles and tradeoffs of existing MVEEs. I emphasize the design choices lifted for GHUMVEE.

In Chapter 3, I provide an overview of the causes of inconsistencies and false positive detections one might encounter when running single-threaded programs in an MVEE. I argue that these false positives result from the use of diversification techniques in the program replicae or from reading input that was not retrieved through the system call interface. In this chapter, I suggest solutions to eliminate or tolerate the false positive detections without compromising the security of the MVEE. This chapter is loosely based on a conference article:

- Stijn Volckaert, Bjorn De Sutter, Tim De Baets, and Koen De Bosschere.
  GHUMVEE. Efficient, effective, and flexible replication.
  In *5th International Symposium on Foundations and Practice of Security (FPS'12)*, pages 261–277. Springer, 2013 [133].

In Chapter 4, I extend the discussion on inconsistencies and false positive detections to multi-threaded programs. I argue that inconsistencies in the execution of multi-threaded replicae result from the natural non-determinism in such replicae and that no existing solution that eliminates non-determinism can be readily integrated into a security-oriented MVEE or the replicae because the existing solutions are limited in scope and/or do not tolerate diversification techniques. I suggest to integrate synchronization replication agents into the replicae and present and evaluate three possible replication strategies and four possible implementations of said agents. Our synchronization agents work even in diversified replicae because they are address-agnostic and do not rely on execution properties that are likely to differ in the different replicae. I further describe practical strategies to implement the replication agents and to embed them into real-world low-level libraries. This chapter is based on a paper submitted to a journal:

- Stijn Volckaert, Per Larsen, Bjorn De Sutter, and Koen De Bosschere.
  Multi-Variant Execution of Parallel Programs.
  Manuscript under review.
  Submitted to *IEEE Transactions on Dependable and Secure Computing (TDSC)* [134].

In Chapter 5, I present "Disjoint Code Layouts" (DCL), a novel and practical diversification technique that offers strong protection against certain memory exploits at a very low cost. DCL guarantees that at any given virtual address, no more than one replica can map a valid executable region. This technique is highly effective against attacks such as ROP, that rely on exact knowledge about the attacked program's memory layout [121]. DCL is easy to deploy and does not rely on intrusive program transformations. This chapter is based on the work we described in a journal article:

- Stijn Volckaert, Bart Coppens, and Bjorn De Sutter.
  Cloning your Gadgets: Complete ROP Attack Immunity with Multi-Variant Execution.
  To appear in *IEEE Transactions on Dependable and Secure Computing (TDSC)* [130].
  DOI:10.1109/TDSC.2015.2411254

In Chapter 6, I introduce the concept of relaxed monitoring and present ReMon, a novel design for a security-oriented MVEE. ReMon combines GHUMVEE with IP-MON, a new component that can be embedded into the replicae. IP-MON can execute system calls and replicate their results, without reporting these system calls to GHUMVEE. We present several possible policies that define the responsibilities of IP-MON and that offer different levels of security and performance. This chapter is based on joint research between the System Software Lab at Ghent University, the Secure Systems Lab at the University of California, Irvine (UCI), and Immunant Inc. I conducted a significant portion of this research while visiting UCI as an intern (Junior Specialist). We described the results of the research in a paper submitted to a conference:

- Stijn Volckaert, Bart Coppens, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz.
  Free Unlimited Calling: Relaxed Multi-Variant Execution.
  Manuscript under review.
  Submitted to *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)* [131].

In Chapter 7 finally, I conclude the dissertation, suggest lines of future work and discuss open issues.

The research I present in this dissertation touches upon several orthogonal research domains. I have therefore chosen to discuss the relevant related work in each chapter, rather than to dedicate a separate chapter to it.

1.3.1 Other Contributions

In addition to the contributions I present in this dissertation, I have also contributed to the following publications during the course of my PhD research:

- Daan Raman, Bjorn De Sutter, Bart Coppens, Stijn Volckaert, Koen De Bosschere, Peter Danhieux, and Erik Van Buggenhout.
  DNS Tunneling for Network Penetration.
  In *Proceedings of the 15th Annual International Conference on Information Security and Cryptology (ICISC'12)*, pages 65–77, Springer, 2013 [107].

- Bert Abrath, Bart Coppens, Stijn Volckaert, and Bjorn De Sutter.
  Obfuscating Windows DLLs.
  In *Proceedings of the 1st IEEE/ACM International Workshop on Software Protection (SPRO'15)*, pages 24–30, IEEE, 2015 [3].

- Stephen Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz.
  It's a TRAP: Table Randomization and Protection against Function-Reuse Attacks.
  To appear at *22nd ACM Conference on Computer and Communications Security (CCS'15)* [36].

The latter paper describes readactor++, a probabilistic defense against function reuse attacks such as COOP and RILC that are not mitigated by current code randomization techniques. This paper is the result of joint research between the Secure Systems Lab at the University of California, Irvine (UCI), the System Software Lab at Ghent University, the Systems Security Lab at the Ruhr-Universität Bochum, and the System Security Lab at the Technische Universität Darmstadt. I conducted a significant portion of this research while visiting UCI as an intern (Junior Specialist).

Chapter 2

Multi-Variant Execution

A Multi-Variant Execution Environment (MVEE) runs several instances of the same program (often referred to as replicae or variants) side by side. The MVEE's main component, a monitor, feeds these replicae the same inputs and monitors their behavior. The replicae are constructed such that they behave identically under normal operating conditions but diverge when under attack. In recent years, over half a dozen systems have been proposed that match the above description. While most of them show many similarities, some authors have made radically different design choices.

2.1 High-Level Design

Broadly speaking, there are two major key factors that distinguish their high-level designs: monitoring granularity and placement in the software stack. In this section, we review these two key factors, point out their implications, and justify the design choices we lifted for GHUMVEE.

2.1.1 Monitoring Granularity

Monitoring the replicae's behavior can be done at many granularities, ranging from monitoring only explicit I/O-operations, to system calls, function calls or even individual instructions. In practice, however, existing MVEEs either monitor at I/O-operation granularity or at system call granularity. Among the MVEEs that monitor at system call granularity, there are some that monitor all system calls, while the others

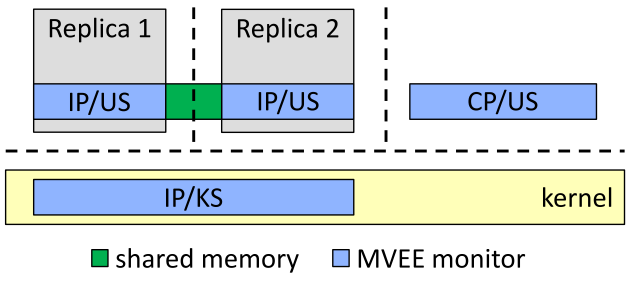


Figure 2.1: Possible placements of an MVEE in the Software Stack

monitor only "sensitive" calls. There is some debate over what the ideal monitoring granularity is. Coarse-grained monitoring yields better performance but might not guarantee the integrity of the system.

Most MVEEs monitor at system call granularity. The reasoning is that on modern operating systems that use page-level memory protection, every application is confined to its own address space. An application must therefore use system calls to interact with the system in any meaningful way. The same holds for exploits. If the ultimate goal of an attack is to compromise the target system, then the attack's payload must invoke system calls to interact with the system.

It makes little sense to monitor at finer granularity levels for the sole purpose of comparing replicae's behavior. The premise of Multi-Variant Execution is that the replicae are constructed such that they react differently to malicious input. While a given malicious input might be sufficient to seize control of one specific replica, it will not have the desired effect on other replicae. These other replicae will either crash or behave differently. As several authors have argued in the literature, both of these outcomes are visible at the system call level [35,114].

2.1.2 Placement in the Software Stack

The placement of the MVEE within the software stack has far-reaching consequences for the MVEE's security and performance properties. This placement is motivated by the conflicting goals of ensuring maximum security and maximum performance. To maximize performance it is of vital importance to minimize the overhead on the interaction between the replicae and the monitor. Since most monitors intervene in every system call invocation, such interactions can occur frequently.

All existing monitors interact synchronously with the replicae. When a replica instigates an interaction with the monitor, it must wait until the monitor returns the control flow to the replica before it may resume its execution. To achieve maximum performance, it therefore is of vital importance to minimize this waiting time, which is dominated by the latency on the monitor-replica interaction. If the monitor runs as a separate process (Cross-Process or CP), then the interaction latency is high because the kernel must perform a context switch to transfer the control from the replica to the monitor. Context switches are notoriously slow as they require a page table and a Translation Lookaside Buffer (TLB) flush [14]. CP monitors can therefore be detrimental for the replicae's performance.

CP monitors are the most interesting choice from the security perspective however. Address spaces form a hardware-enforced boundary between processes. Placing the monitor outside the replicae's address spaces therefore protects it from misbehaving replicae. Table 2.1 illustrates that most authors recognize the importance of such a hardware enforced boundary. Almost all of the existing monitors prioritize security over performance and run Cross-Process. These monitors correspond with the label "CP/US" in Figure 2.1. N-Variant Systems is a notable exception [35]. The N-Variant monitor runs within the same address space as the replicae (In-Process or IP), but it is protected from them misbehaving by the kernel-user space boundary. This design is represented by "IP/KS" in Figure 2.1. This is, at least in principle, the ideal approach. However, it does have the downside of enlarging the Trusted Computing Base (TCB). This is undesirable from a security standpoint [111].

VARAN finally implements a third design that is represented by "IP/US" in Figure 2.1 [53]. VARAN is a reliability-oriented IP monitor, embedded into the replicae. It therefore consists of several components, each of which can communicate directly with the replica in which it is embedded. VARAN primarily intends to increase the reliability of the replicae when, e.g., testing new patches. It therefore uses a less secure design than the aforementioned "CP/US" MVEEs. VARAN's authors also recognize this fact.

GHUMVEE is a security-oriented MVEE and is therefore implemented as a CP/US MVEE. In Chapter 6 we will also propose a hybrid design called ReMon. ReMon is based on GHUMVEE, but it also includes an In-Process component, which makes ReMon a hybrid

	User-Space	Kernel-Space
In-Process	VARAN [53]	N-Variant Systems [35]
Cross-Process	DieHard [18], Cavallaro [27], Orchestra [114], Tachyon [85], Mx [52], GHUMVEE [133]	

Table 2.1: Classification of existing MVEEs based on their position in the software stack.

CP+IP/US MVEE.

2.2 Basic operation of GHUMVEE

GHUMVEE launches its replicae by forking them off its main thread and by executing a sys_execve system call in the context of the forked off processes. Prior to this call, the newly created replica processes establish a link between GHUMVEE's monitor and themselves by requesting to be placed under a monitor's supervision after which they raise a SIGSTOP signal. The kernel suspends the replicae after they have raised this signal and it reports their status to the monitor. The monitor can then resume the replicae and begin to monitor their execution.

2.2.1 Monitoring System Calls

Like most MVEEs, GHUMVEE monitors the replicae's behavior at the system call interface by intervening at the call and return site of every system call. In theory, different mechanisms are available to implement system call interception. In practice however, the monitor's place in the software stack dictates which mechanism must be used. MVEEs that run Cross-Process and in User-Space rely on the operating system's debugging infrastructure to place the replicae under their control and to intercept their system calls.

Each UNIX-based operating system offers a debugging API that allows monitors to intervene at the start and return of every system call. Security-oriented monitors typically leverage this API to run the replicae in **lock-step** at the system call level. The monitor suspends each replica that enters or exits from a system call until all replicae have reached the same call or return site. At this point, the replicae are said to have reached a **Rendez-Vous Point (RVP)** (sometimes referred to as a synchronization point).

The monitor asserts that the replicae are in equivalent states whenever they reach such a RVP by comparing the system call arguments. Two sets of system call arguments are considered to be equivalent if they are identical (in case of non-pointer arguments) or if the data they refer to is identical (in case of pointer arguments). Salamat gives a formal definition of equivalent states [113].

If the replicae are not in equivalent states at a RVP, the monitor raises an alarm and takes the appropriate action. GHUMVEE will consider all tasks that share an address space with one of the replicae that caused the discrepancy as tainted, and it will therefore terminate these tasks. Do note that this will not necessarily stop the entire program since a program might consist of several tasks that do not share address spaces.

Reliability-oriented monitors that are, e.g., used to test new software patches may differ from **security-oriented** monitors, such as ours, with respect to system call monitoring. VARAN for example does not enforce **lock-step** execution [53]. Instead, it lets the master replica run ahead of the slave replicae and it caches the arguments and results of all of the master replica's system calls so that they may be consulted by the slave replicae at a later point.

2.2.2 Transparent Execution

Many system calls require special handling to ensure that the Multi-Variant Execution is transparent to the end-user. With the exception of run-time overhead, the end-user should not be able to notice that more than one replica of the program is running. GHUMVEE therefore uses a master/slave replication model. One of the replicae is the designated master replica and the other replicae are slaves. GHUMVEE ensures that only the master replica can execute system calls that have visible effects on the rest of the operating system. Specifically, these are the system calls that correspond with I/O operations. Whenever the replicae reach a RVP at the start of an I/O-related system call, GHUMVEE will verify that the replicae are in equivalent states, and then overwrite the system call number in the slave replicae by that of a system call with no visible effects. GHUMVEE currently uses sys_getpid for this purpose since that is a trivial and fast system call. When GHUMVEE subse-

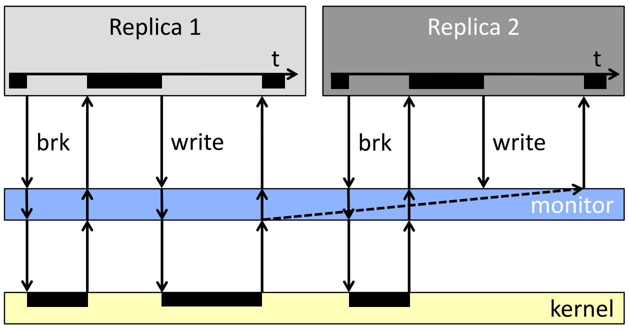


Figure 2.2: Transparently executing I/O-related system calls

quently resumes all replicae, only the master replica will execute the intended I/O operation.

At the next RVP, when all replicae have returned from their system call, GHUMVEE will copy the results of the system call from the address space of the master to the address space of the slave replicae. We refer to this mechanism as **master calls**. System calls that do not require special handling, other than consistency checking, and that may therefore be executed by **all** replicae are called **normal calls**.

Orchestra uses a different mechanism to ensure transparent execution. Orchestra's monitor executes all I/O-related system calls on behalf of the replicae and copies the system call results from the monitor to the replicae. Salamat et al. refer to this mechanism as **monitor calls** [114]. This mechanism has an interesting implication. Since the replicae never perform I/O operations themselves, they do not need to open or close any file descriptors[1]. This design allows replicae running on top of Orchestra to continue executing even if one of the replicae is shut down after being compromised. GHUMVEE, by contrast, is forced to shut down all of its replicae if the master gets compromised, because neither GHUMVEE's monitor, nor any of the slave replicae open any file descriptors.

2.2.3 Injecting System Calls and Restarting Replicae

On top of the above tasks, GHUMVEE can also inject new system calls and, as a result, rewind replicae to their initial state. Injecting system calls can be useful to transparently add new functionality to the

[1]We discuss one exception to this rule in Section 3.3.

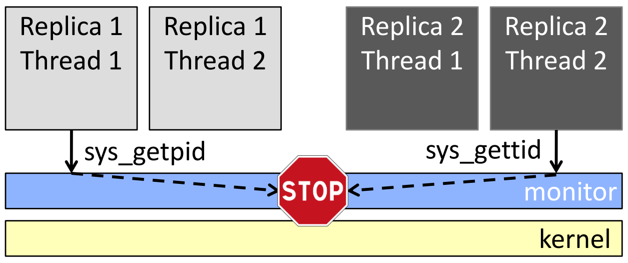


Figure 2.3: False positive detection in a multi-threaded program

replicae. To inject a system call in a replica, GHUMVEE waits until the replica has reached a RVP. At this point, GHUMVEE will store a backup of the register context of the replica and it will overwrite the system call arguments.

Many system calls accept arguments that are stored in data buffers. To inject such arguments, GHUMVEE searches for a writable memory page in the replica that is large enough for the arguments. If the replica is multi-threaded, GHUMVEE will search for the replica's thread-local stack, in order not to corrupt memory that might be used by other tasks that share an address space with the replica.

GHUMVEE then reads and stores the original content of the memory page and writes the arguments into that page. GHUMVEE will then resume the replica and wait until the injected system call returns. At that point, GHUMVEE will restore the original contents of the overwritten memory page and restore the original register context, prior to the system call injection.

Restarting replicae to their initial state is a trivial extension of this system. To support restarting, GHUMVEE stores the original arguments of the sys_execve call that was used to start the replica, as well as the environment variables [46] at the time of the original sys_execve invocation. Whenever a replica reaches a RVP, GHUMVEE can restore the original environment variables and inject a new sys_execve call with those original arguments using the mechanism described above to restart the replica.

GHUMVEE uses this restart mechanism to enforce "Disjoint Code Layouts", as we will explain in Chapter 5.

2.2.4 Multi-Threaded Monitor

Replicae that consist of multiple tasks, whether they be processes or threads, can trigger false positive detections in an MVEE. One of the reasons for these false positives is that an MVEE cannot control the order in which its replicae are scheduled. At best it can slightly manipulate the scheduler by artificially stalling the replicae. Consider for example a program that has one thread that calls sys_getpid in a tight loop and one thread that calls sys_gettid in a tight loop. If we run two replicae of this program side by side, and if in one replica the first thread gets scheduled first, while in the second replica the second thread gets scheduled first, a naive monitor might detect a mismatch because the first replica will execute sys_getpid first, while the second will execute sys_gettid.

The obvious solution to this problem is to compare the behavior of equivalent threads, rather than the behavior of the replica as a whole. A second problem that can easily be addressed is that a single monitor instance quickly becomes a bottleneck when monitoring replicae that consist of multiple tasks. We therefore create additional instances of the monitor, one for each set of equivalent tasks, and have each of these instances monitor only that set of tasks. GHUMVEE's mechanism for creating additional instances of the monitor and for attaching these instances to the appropriate tasks is almost identical to the one that was described by Salamat [113].

The last and most fundamental problem with multi-task replicae is that such replicae often do not execute deterministically because these tasks might communicate internally over shared memory. Hosek suggested to tolerate small variations in the replicae's behavior to relieve this problem [53]. During the course of my research however, I have encountered many programs whose non-deterministic execution results in behavioral discrepancies that are practically indistinguishable from actual attacks.

In Chapter 4 I present a detailed overview of this problem and propose solutions to impose deterministic execution on replicae with minimal overhead.

2.3 Monitoring and Replication Infrastructure

Monitoring utilities, like GHUMVEE, that run outside the context of the process they monitor and in user-space are traditionally implemented using the operating system's debugging facilities. GNU/Linux offers access to its debugging facilities through the process_vm_readv [73], wait4 [79] and the multi-purpose ptrace API [74]. At the time of this writing, the ptrace API implements 31 commands. In this section, I present an overview of the most relevant commands and describe how and why they are used by GHUMVEE.

2.3.1 Attaching to the Replicae

The ptrace API offers the ability for one process, often referred to as the debugger, to place another process, often referred to as the debuggee, under its supervision. Linux, and other operating systems that offer a similar API, allows for one debugger per debuggee. However, a single debugger may attach itself to many debuggees.

Both processes can instigate this link. The debuggee may use the PTRACE_TRACEME command to place itself under its parent's supervision. The debugger on the other hand may use the PTRACE_ATTACH or PTRACE_SEIZE commands to establish the link. After the link has been established, the debugger gains access to many privileged operations.

2.3.2 Transferring Replicae

The debugger can configure its link with the debuggee such that all processes and threads that are created by the debuggee are also placed under the debugger's supervision. However, as I pointed out in Section 2.2.4, GHUMVEE uses a separate monitor thread for each set of equivalent tasks. New monitor threads do not have the privilege to monitor tasks that they have not attached to, even though the creator of the monitor thread may have this privilege. The underlying reason for this phenomenon is that the Linux kernel uses the debugger's task id to authenticate ptrace operations. Threads do not share task ids, even though they might be in the same process. Therefore, whenever a replica creates a new process or thread, the monitor thread under whose supervision it was placed must detach itself from the new process or thread first. The monitor thread can detach itself using the

PTRACE_DETACH command. Then, the newly created monitor thread can attach itself to the new process or thread.

Transferring debuggees from one debugger to another is a cumbersome process because the kernel allows the debuggee to run freely and unsupervised when their debugger exits or detaches from the debuggee. The debugger therefore has to take the necessary precautions to ensure that the debuggee cannot run uncontrolled.

Salamat proposed to let newly created tasks run until they hit their first system call, and to replace the call number of this first system call by that of sys_pause. When the monitor then detaches from the task, the task would simply invoke the sys_pause call, which causes it to wait until a signal is received. The newly created monitor thread could then safely attach to the task.

GHUMVEE has a slightly different solution for this problem. GHUMVEE overrides the entry point of each newly created task by the address of a small, infinite loop. The original monitor can therefore detach from the newly created tasks as soon as they are created and the new monitor thread simply needs to restore the original entry point.

2.3.3 Stopping and Resuming Replicae

From a debugger's point of view, a debuggee is either running or in one of the *ptrace-stopped* states. Most of the ptrace commands require that the debuggee is in a *ptrace-stopped* state and the debuggee will automatically enter such a state when it has triggered an event that requires the debugger's attention.

After an event has been reported to the debugger, it can resume the execution of the debuggee using one of several ptrace commands. The most interesting among these commands are PTRACE_CONT and PTRACE_SYSCALL. PTRACE_CONT resumes the debuggee until it is interrupted by the delivery of a signal or until its process group is stopped. PTRACE_SYSCALL is similar to PTRACE_CONT but it will also stop the debugger when it enters or exits from a system call. This command is particularly useful for a monitor because it allows the monitor to implement RVPs.

A debugger must poll the kernel using one of the "wait" system calls to receive notifications for its debuggees [79]. Process state changes such as transitions from the running to a *ptrace-stopped* state can only be reported through one of the "wait" calls. The "wait" calls can be used in blocking or non-blocking mode. When used in blocking mode, the call will not return until it has received an event. When used in non-blocking mode, the call will return immediately, even if no event has been reported.

GHUMVEE uses the sys_wait4 call to poll the kernel for *ptrace-stopped* events because it is slightly faster than the other "wait" calls. The "wait" calls are not ideally suited for use by a multi-threaded monitor because it does not offer fine-grained selection of the tasks whose state the monitor is interested in. Specifically, the "wait" calls allow the monitor to either receive notifications for (i) all processes in a given process group, (ii) for one specific process with a given process id, or (iii) for all of the monitor's child processes.

In an MVEE, option (i) can only be used in non-blocking mode because each replica is in its own process group. A monitor can therefore not use option (i) in blocking mode because it only allows the monitor to receive events from one of its replicae. Using option (i) in non-blocking mode is not ideal either because it forces the monitor to either unnecessarily consume CPU cycles, or to periodically relinquish the CPU, which in turn increases the monitor's latency in reacting to events.

By the same reasoning, option (ii) can also only be used in non-blocking mode. Each monitor is responsible for monitoring a set of equivalent tasks, and it must therefore issue several "wait" calls to receive events for each of these tasks.

Option (iii) can be used in blocking mode, but not by the monitor's main thread. In Linux, all processes that are forked off another thread are considered to be that thread's children. The tasks that a thread is attached to using ptrace are also considered to be that thread's children.

As we described before, GHUMVEE starts its replicae by forking them off the main monitor thread. All of the processes and threads in all of the replicae are therefore considered to be the children of the main monitor thread. If the main monitor thread were to use option (iii) then it could receive notifications for tasks that it is not responsible for and it would have to forward these notifications to the appropriate monitor thread. This would add an unnecessary indirection, unneeded complexity and additional latency. GHUMVEE therefore does not use its main thread for monitoring at all.

In all other threads, GHUMVEE does use option (iii) in blocking mode. This is the ideal design for an MVEE in terms of complexity and latency.

2.3.4 Monitor-Replica Data Transfers

The monitor often needs to read data from or write data to the address spaces of the replicae in order to compare system call arguments and in order to replicate the return values of master calls. Since the monitor and the replicae run as separate processes, this data cannot be transferred directly. Instead, one of the Inter-Process Communication (IPC) mechanisms or debugging APIs must be used.

A first API that is available for data transfers between a debugger and its debuggee is ptrace. ptrace offers the PTRACE_PEEKDATA and PTRACE_POKEDATA commands that can copy a data block from the debuggee to the debugger and vice versa. The size of the data blocks that can be copied using these commands is fixed to one memory word, i.e., 4 bytes on 32-bit systems and 8 bytes on 64-bit systems. This is a severe limitation since many of the data blocks that are typically transferred between the monitor and its replicae are much bigger than one memory word.

Shared memory can be used as an alternative to ptrace. The monitor can allocate a block of shared memory using the System V IPC API [76] or the sys_mmap system call. This block can be mapped into a replica's address space transparently by means of injected system calls. All monitor-replica data transfers can from then on be routed through this shared block. To do so, the monitor must force the replica to copy the data from its original location to the shared block, or vice versa, depending on the direction of the transfer. One way to do this is to inject calls to a specially crafted, position-independent memcpy routine. Figure 2.4 illustrates this process. In this figure, the replica invokes a system call at time **t1**. The monitor injects a call to the memcpy routine at time **t2** by overwriting the replica's program counter. The memcpy arguments can be injected by overwriting the values in the rdi, rsi, and rdx on 64-bit x86 systems or by writing them on the replica's stack and overwriting the value in the esp register on 32-bit x86 systems. After injecting the memcpy call, the monitor returns the control to the replica. The replica copies the system call arguments into the shared block at time **t3** and invokes another system call to indicate the end of the memcpy operation. The control then returns to the monitor at time **t4**. After reading the system call arguments from the buffer and comparing them with the other replicae's arguments, the monitor restores the original register values at time **t5** and resumes the original system call.

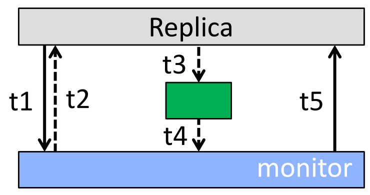


Figure 2.4: Routing monitor-replica data transfers through a shared memory block.

For GHUMVEE, we originally opted not to use the shared memory method. Instead, we designed and implemented a kernel patch that extends the ptrace API. We added two new ptrace commands. PTRACE_EXT_COPYMEM can copy a fixed size data block between a debugger and a debuggee or between two debuggees that are being controlled by the same debugger. Similarly, PTRACE_EXT_COPYSTRING can copy a NULL-terminated C string.

Currently however, GHUMVEE uses the process_vm_readv API [73] for data transfers. This API was introduced in Linux 3.2 and yields similar performance to PTRACE_EXT_COPYMEM. It does have two minor disadvantages. First, it cannot be used to copy between two debuggees directly. The debugger's address space must be used as an intermediate. Second, it cannot be used to copy C strings because their size is often not known by the debugger.

We compared these four data transfer methods by running synthetic benchmark programs on a 32-bit x86 system that runs the Linux 3.16 kernel[2]. The benchmark programs first allocate a large buffer of 200Mb and then perform one million system calls in a tight loop. In this first benchmark, whose results are illustrated in Figure 2.5, we used a system call that accepts a fixed sized data buffer as its argument. For the second benchmark, whose results are illustrated in Figure 2.6, we used a system call that accepts a NULL-terminated C string as its argument. Prior to each system call invocation, both benchmark programs write their system call argument into a randomly selected region within the large buffer. The size of the system call argument can be passed to the program as a command line parameter.

[2]The experiment yielded similar conclusions on a 64-bit system. GHUMVEE does not support the shared memory method on 64-bit systems though.

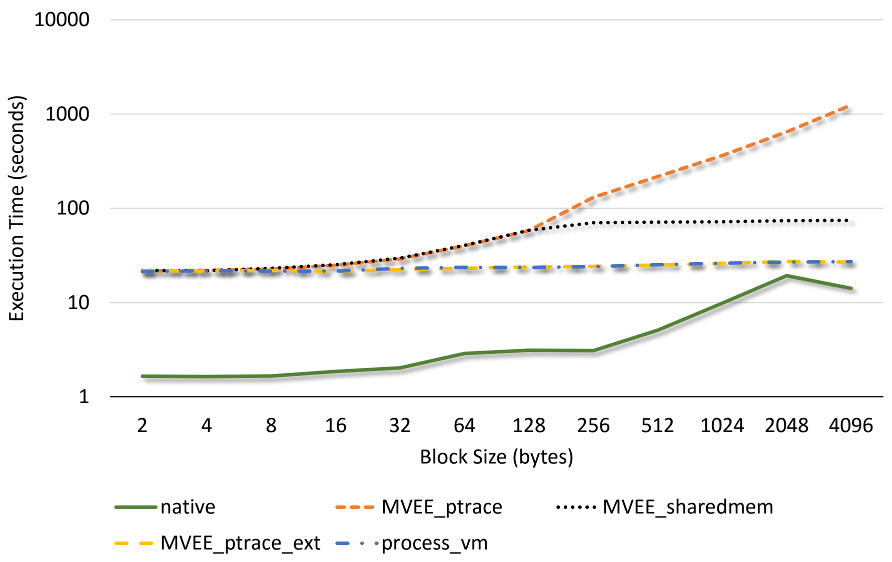


Figure 2.5: Comparison of the performance of different monitor-replica data transfer methods for fixed sized data blocks.

When running this benchmark program in GHUMVEE, the monitor copies the system call argument from the replicae's address spaces to its own address space to compare them. In each run, the monitor therefore performs approximately two million data transfers.

We measured the performance of the data transfer methods for several argument sizes by calculating the average run time of five iterations of each benchmark with two replicae. We disabled Address Space Layout Randomization (ASLR), hyper-threading and all forms of dynamic frequency and voltage scaling during the measurements to maximize the reproducibility of the results. Furthermore, we always seeded the random-number generator we used to randomly select the region with the same seed value.

As we can see in Figure 2.5, the official process_vm_readv has rendered GHUMVEE's PTRACE_EXT_COPYMEM extension void. Figure 2.6, however, shows that the PTRACE_EXT_COPYSTRING extension is still the fastest method for inter-process string copying but such operations are rarely neccessary since only a handful of system calls accept C strings as arguments. We can therefore no longer justify patching the kernel with GHUMVEE's ptrace extensions.

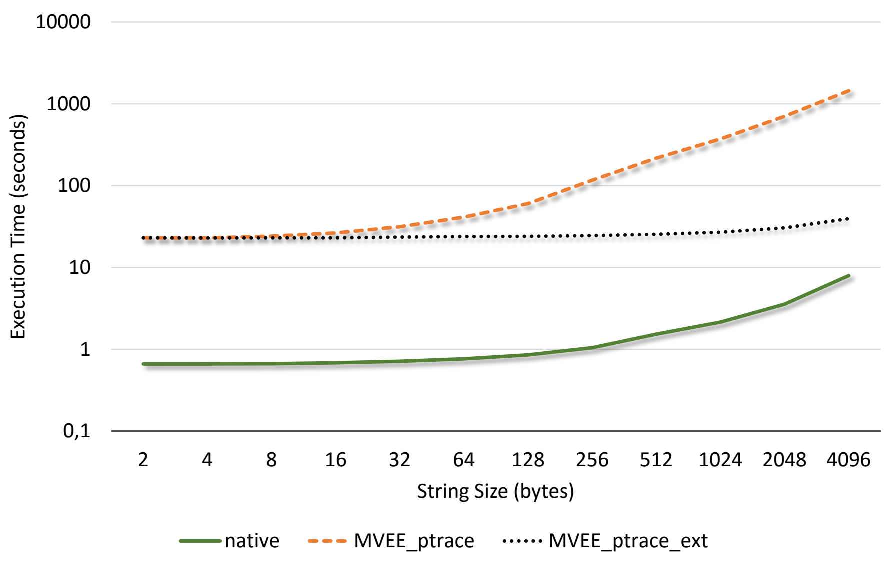


Figure 2.6: Comparison of the performance of different monitor-replica data transfer methods for NULL-terminated C strings.

2.3.5 Performance Implications

Modern operating system kernels are constructed such that native programs can perform most system calls without having to pay the cost of context switching. In such cases, the program only incurs modest performance losses due to mode switching. This is however no longer the case when a debugger is attached to the program. A debugger and its debuggee run in separate processes and are therefore separated by a hardware-enforced boundary. Consequently, the kernel must switch to the context of the debugger whenever the debuggee triggers an event that requires the debugger's attention. Context switching is costly. Besides being a computationally intensive task, it also involves page table switching and TLB flushes [14].

Naively, one could avoid such flushes by preventing the debugger from running on the same CPU core as the debuggee. The kernel can then preserve the contents of the page table register on the debuggee's CPU core. The down side of this approach is that all information that is relevant to the events the debuggee triggers must now propagate through the CPU cache and/or the memory bus to the debugger's CPU core. We can therefore see a trade-off between cache and memory pressure on the one hand and TLB pressure on the other hand.

We have constructed two test cases to gain insight in this trade-off. We tested our test cases on an Ubuntu 14.04 x64 machine with two Intel Xeon E5-2600 CPUs with eight physical cores each. Dynamic frequency and voltage scaling features, as well as hyper-threading were disabled during our tests. We used the Linux 3.16 kernel for these tests.

For our first test case, we built a program that runs five million `sys_getpid` system calls in a tight loop. `sys_getpid` is a typical example of a system call that can be handled without having to switch contexts. This program is therefore very fast. When running natively on our testing machine, it runs in 0.3689 seconds on average when measured over 10 runs. We then ran one replica of this program under the control of a simple MVEE.

In our first test, we did not force the MVEE's monitor or the program to run on specific CPU cores. The MVEE's monitor does little more than stopping the program at every system call entrance and exit, reading the system call number at the system call entrance and the return value at the system call exit, and resuming the program immediately. Under these conditions, the program runs in 83.89 seconds on average, or over **227x** slower than the original program.

A second test yielded marginally better results. In this test, we used the `sched_setaffinity` API to force the MVEE's monitor and the program to run on separate CPU cores. This approach minimizes the amount of additional TLB misses we introduce in the program, but does increase the pressure on the CPU cache and memory subsystem. The program now ran in 82.32 seconds on average, or **223x** slower than the original program.

In our third test, we forced the MVEE's monitor and the program to run on the same CPU core. This approach minimizes the additional pressure on the memory subsystem but does incur the maximum amount of additional TLB pressure. Nonetheless, this method yielded significantly better results. The program now ran in 36.98 seconds in total, or **100x** slower than the original program.

For our second test case, we used the `md5sum` program to calculate a file hash for a 1.2GB disk image. Natively, this program ran in 3.81 seconds on average and invoked 40715 system calls in total. We then ran two replicae of this program under the control of our MVEE and tested the same scenarios. In this case, our MVEE performed tasks that were not necessary in the previous case. Specifically, our MVEE verified the equivalence of all system calls and replicated the input the program receives from the `sys_read` calls to both replicae. For both of these tasks, the MVEE transfers data between the address spaces of the replicae and its own address space. These data transfers obviously introduce additional cache and memory pressure.

In the first test, with neither of the replicae, nor the monitor itself forced to run on specific cores, the program ran in 4.65 seconds on average, or **1.22x** slower than the original program.

In the second test, with the replicae and the monitor all forced to run on different cores, the program ran in 4.62 seconds on average, or **1.21x** slower than the original program.

In the third test, we forced the monitor and the master replica to run on one core and the slave replica to run on another. The program now ran in 4.51 seconds on average, or **1.18x** slower than the original program.

These tests show that the overhead an MVEE incurs when intercepting system calls is significant. ReMon, the hybrid IP+CP/US MVEE we present in Chapter 6, has a much lower performance impact than GHUMVEE because it can monitor and replicate innocuous system calls in its In-Process component, thus avoiding the overhead of the Cross-Process component.

2.4 Debugging Features in GHUMVEE

The Linux kernel does not allow for multiple debuggers to attach to the same debuggee using the ptrace API. This design choice is understandable, but very unfortunate in our context since GHUMVEE uses the ptrace API to monitor the replicae, and since being able to debug replicae while they are being monitored can be tremendously useful.

To solve this problem, we have equipped GHUMVEE with much of the same debugging features one would typically find in debugging tools such as gdb and strace:

- GHUMVEE can be configured to log event traces for all of its replicae. These event traces include information regarding the signals that are sent to each replica as well as the system call numbers, names, and arguments for all of the system calls each replica performs. They also include timestamps for each event. GHUMVEE can optionally also log complex system call arguments such as I/O and message vectors.

- GHUMVEE supports full backtracing. It exposes a fairly straightforward API that can be used to log the full call stack for each variant. This call stack includes the callee addresses, as well as the function names, source files and line numbers associated with each callee. Optionally, it can also include a dump of the full register context. At the heart of this API lies a fully-featured C++ class that can be used to parse debugging information in the DWARF2/3 format. Contrary to popular libraries such as libbacktrace, GHUMVEE's class for debug information parsing can parse information from exception handler frames as well as debug frames.

- One can instruct GHUMVEE to generate backtraces by sending a SIGQUIT signal to the main monitor thread.

- GHUMVEE can set soft- and hardware breakpoints on code and data addresses in the replicae. When a breakpoint is hit, GHUMVEE logs a backtrace for the replica and resumes its execution.

- GHUMVEE can log the contents of the synchronization buffer (cfr. Chapter 4) as well as the IP-MON buffer (cfr. Chapter 6) in a human-readable format.

This functionality has proven useful while, e.g., developing new synchronization replication agents (cfr. Chapter 4). In the future, it can also serve as a base to extend GHUMVEE with twin or delta debugging functionality (cfr. Chapter 7).

2.5 Comparison with other MVEEs

We compare GHUMVEE with other MVEEs in Table 2.2. This table shows high-level properties and design choices, which we reviewed in this chapter. It also summarizes which inconsistencies each MVEE can deal with when executing single-threaded replicae. We discuss these inconsistencies in-depth in Chapter 3. We also recap each MVEE's capabilities for supporting multi-threaded replicae. This will be the subject of Chapter 4.

	N-Variant Systems	Cavallaro	Orchestra	Mx	Tachyon	VARAN	GHUMVEE
general properties							
open source	yes	no	no	no	no	no	yes
intent	security	security	security	reliability	reliability	reliability	security
RVPs	system calls	system calls	system calls	system calls	system calls	system calls	system calls + functions
monitor type	IP/KS	CP/US	CP/US	CP/US	CP/US	IP/US	CP/US
architecture	i386	i386	i386	x86-64	x86-64	x86-64	i386 + x86-64
system calls supported	130	?	~50	?	?	86	198
tested on	Apache	thttpd	SPEC CPU 2000, Apache, Snort	SPEC CPU 2006, coreutils, redis, lighttpd	cURL, mplayer, php5, ncompress, htget, gs, glftpd, socat, corehttp, compress, primegaps, mencoder, lighttpd, thttp, coreutils	SPEC CPU 2000, SPEC CPU 2006, Apache, thttpd, lighttpd, memcached, redis, beanstalkd	SPEC CPU 2006, PARSEC 2.1, PHORONIX 4.8, Apache, thttpd, lighttpd, memcached, redis, beanstalkd, mplayer, firefox, libreoffice, GNOME/KDE desktop apps
inconsistencies dealt with							
signals	not supported	not supported	optimized delivery	?	not supported	delivered instantly	delivered at system call RVPs
system call interruption due to signals	not supported	not supported	not supported	?	not supported	supported	supported
RDTSC interception	no	no	no	no	no	no	yes
vdso system calls intercepted	no	no	no	no	no	yes	yes
support for parallelism							
fork/exec	not supported	supported	supported	supported	supported	supported	supported
multithreading	limited support	limited support	limited support	limited support	limited support	limited support	full support
synchronization replication	no	no	no	no	no	no	yes
support for diversity							
implicit input replication	not supported	not supported	not supported	not supported	not supported	not supported	limited support
diversification techniques	address space partitioning, instruction set tagging	address space partitioning	reverse stack growth, system call number randomization	multiple program revisions	multiple program revisions	multiple program revisions	full ASLR, disjoint code layouts
transparency and ease of use							
kernel patch required	yes	no	no	no	yes	no	optional
debugging	limited system call logging	?	no	?	?	?	full event logging, backtracing, breakpoints, buffer logging

Table 2.2: Comparison of GHUMVEE with other MVEEs.

Chapter 3

Inconsistencies and False Positive Detections

The MVEE must feed all replicae the same input in order to guarantee that they will behave identically under normal operating conditions. For explicit input operations, such as reading an incoming packet from a socket, the monitor can satisfy this requirement by applying the **master call** mechanism we described in Section 2.2.2 to system calls such as sys_read.

In some cases this is not sufficient, however. Several sources of input can be accessed directly, without invoking any system calls. The replicae often behave differently after reading input from such sources. This can lead to false positive detections by the monitor. In this chapter, we summarize the sources of input that can be accessed directly and describe how we provide consistent input from such sources to all replicae.

3.1 Shared Memory

All commodity operating system kernels offer a **file mapping** API and an **Inter-Process Communication (IPC)** API that can be used to share physical memory pages among several processes.

The file mapping API, which can be accessed through the sys_mmap system call on Linux systems, allows for programmers to associate individual physical memory pages with regions within a file on the file system. The associated file is often referred to as the **backing file**. When a page fault is triggered on a physical page that is backed by a file, which happens when this page is accessed for the first time, the operating system will load the contents for the page from the associated region in the backing file. The operating system will also write the contents of the page back to the file should the page ever become dirty.

The programmer can specify which region of the backing file each memory page corresponds to and whether or not the changes should be written back to the file. However, even if the programmer requests that changes be written back to the file, the operating system will only do so if the programmer has opened the backing file with read/write access. For some backing files, such as system libraries, the operating system will deny any requests made by a non-priviliged user to open the file with read/write access and will instead only allow read access.

Programmers often use file mapping as an efficient way to access files. A mapped file can be accessed directly, without having to invoke sys_read or sys_write calls. The file mapping API is also commonly used to create shared memory pages. A program can create a temporary file with read/write access and map this temporary file into its own address space. Other programs can then map the same file into their address spaces, thus sharing the associated physical memory pages with the program that created the file.

Programmers can also use the IPC API, which can be accessed through the sys_ipc or sys_shmget/sys_shmat system calls on Linux systems, to create and map shared physical memory pages not associated with a backing file. These pages have a unique identifier. Programs that know this unique identifier can map the associated physical pages into their virtual address spaces.

Shared memory pages often constitute a problem within an MVEE. Replicae can read from shared memory pages without invoking a system call and, consequently, are not subject to the lock-step execution mechanism we discussed in Section 2.2.1 when doing so. The MVEE's monitor can therefore not guarantee that the replicae will read the same input from shared memory pages that are being written to by an external process. Similarly, the replicae could also write to the pages directly, which prevents the MVEE's monitor from asserting that the replicae write the same data to the pages.

A possible solution to this problem is to revoke the replicae' access rights to all shared memory pages. Each read from or write to the shared pages would then result in a page fault. The operating system would translate this page fault into a SIGSEGV signal, which is normally passed down to the program so it can invoke its signal handler. When a debugger is attached, however, a notification is sent to the debugger first and the actual signal is not passed to the program until the debugger has approved it. In an MVEE, this mechanism could be used to intercept all accesses to shared memory. For each SIGSEGV signal that results from a read operation on a shared memory page, the monitor could perform the read operation itself and replicate the results to all replicae. For write operations, the monitor could perform the write itself. The monitor could then prevent the SIGSEGV signal from being delivered, thus effectively emulating all accesses to the shared memory pages. Emulating accesses to shared memory is unfortunately prohibitively slow [83] and completely negates the performance benefits of using shared memory in the first place.

In GHUMVEE, we therefore opted to deny all requests to map shared memory, unless the monitor can assert that the accesses to the shared memory will not result in inconsistencies. Specifically, GHUMVEE denies all requests to map shared memory through the System V IPC API since any pages mapped through this API can always be written by external processes that know the page identifiers.

For file mappings on the other hand, GHUMVEE does allow read-only shared mappings that are backed by files to which the user does not have write access. Such mappings will have content that is completely static (i.e., the pages cannot be written to by either the replicae or any external process that runs at the same privilege level). The monitor can therefore still guarantee that the replicae will receive the same input. Allowing read-only shared mappings is necessary to support dynamically linked programs since the program interpreter's[1] preferred method of loading shared libraries is by mapping them using the file mapping API.

GHUMVEE does not allow read/write shared mappings. The monitor generally returns an EPERM error when a replica attempts to establish such a mapping, thus indicating that the mapping is not allowed. In specific cases, however, read/write shared mappings are not used to communicate with external processes, but instead simply as an efficient way to access files. To handle these cases, we implemented a **mapping-type-override method**. With this method, GHUMVEE changes the mapping type from shared to private by overriding the arguments of

[1]The program interpreter is an OS component that is responsible for loading programs and setting up their initial virtual address space.

the sys_mmap call that is used to set up the mapping. Private mappings are implemented using copy-on-write (COW) paging. The operating system will therefore create a private copy of the privately mapped page when the replica attempts to write to it for the first time. From that point onwards, external processes can no longer influence the contents of the privately mapped page, which eliminates the need for the monitor to replicate the contents of the pages to all replicae. The monitor does, however, still verify whether the replicae all write the same contents to the privately mapped pages by comparing the page contents when they are unmapped. If the contents of the pages do not match, then the monitor will raise an alarm. If they do match, however, the monitor will write the contents back to the backing file.

3.1.1 Evaluation and Comparison with other MVEEs

The aforementioned method of overriding the mapping type for file-backed shared memory was neccessary to support programs in the KDE desktop suite[2]. These programs use file-backed shared memory to read and write configuration files efficiently. Our method did not cause noticeable slowdowns when running such programs.

Our decision to disallow read/write shared mappings and the use of the System V IPC API does not constitute a big problem either in commodity applications. While shared memory is the preferred method for graphical applications to communicate with the display server, we have not seen a single applications that did not have a fall-back method in place when GHUMVEE's monitor rejected the request to map shared memory pages. This fallback method typically yields significantly worse performance, but is still acceptable in many situations. The MPlayer[3] media player for example also relies on shared memory for hardware-accelerated playback of movies. When running MPlayer in GHUMVEE, it falls back to software-rendered playback. In our tests, we could fluently play back a 1080p h264-encoded movie in MPlayer with a frame drop rate of less than 1%, while running two replicae side by side.

GHUMVEE's handling of shared memory is similar to Cavallaro's MVEE [27], but is more advanced than other security-oriented MVEEs because those do not support the mapping-type-override method.

[2]https://www.kde.org
[3]https://www.mplayerhq.hu/

3.2 Timing Information

Interactive and real-time applications frequently need to measure the length of a time interval to guarantee that they will function correctly. Media players for example need to know exactly when to start rendering a frame. The timing information that these applications rely on must be accurate and precise and must be accessible with minimal overhead. Both processor vendors and kernel programmers therefore offer an interface to access timing information with minimal overhead.

All x86 processors since the original Pentium therefore offer a **ReaD TimeStamp Counter (RDTSC)** instruction which reads the value of a special-purpose register that counts the number of clock cycles since the processor was powered on [58]. This number can be divided by the clock frequency to accurately measure the length of a time interval.

The 64-bit x86 version of the Linux kernel, as well as recent versions of the 32-bit x86 kernel, implement the **Virtual Dynamic Shared Object (VDSO)** [78]. The VDSO is a small dynamically linked library that is mapped into every running program's virtual address space. It consists of two memory pages: an executable memory page that contains code, and a read-only memory page that contains timing information. The VDSO implements virtual system call functions. Each virtual system call is an optimized version of one of the system calls that is exposed by the kernel. Rather than the system call they correspond to, however, the virtual system calls execute entirely in user space, thus avoiding the often costly mode and/or context switches that come with the execution of a normal system call. Linux currently offers virtual system calls for each API that exposes timing information.

Both the RDTSC instruction and the VDSO are therefore sources of timing information that can be accessed without invoking an actual system call. Once again, an MVEE's monitor can therefore not guarantee that replicae that access this timing information will receive consistent input.

GHUMVEE implements workarounds for both problems. GHUMVEE's monitor sets the **Time Stamp Disable (TSD)** flag in the CR4 register of the processor within the context of each running replica [58]. Setting this flag discards the replicae' privilege to execute the RDTSC instruction. Whenever the replica tries to execute an RDTSC instruction, the processor will raise a general protection fault. The operating system will translate this fault into a SIGSEGV signal and will notify the monitor accordingly. Whenever the monitor receives such a notification, it will disassemble the instruction that caused the fault. If the instruction is indeed an RDTSC, then GHUMVEE will execute the instruction on the replicae' behalf and replicate the results.

To eliminate the inconsistencies caused by the VDSO, GHUMVEE will override the arguments of each sys_execve system call. This call is used to execute a program. GHUMVEE changes the name of the program that must be executed into the name of a small loader program we have created. This small loader program, which we aptly called the **GHUMVEE Program Loader (GPL)**[4], deletes the ELF auxiliary vector entry argument that specifies the location of the VDSO [72]. Afterwards, GPL will manually map the original program into the virtual address space, set up the initial stack exactly as it would have been set up had GHUMVEE not overridden the arguments of the sys_execve call, and pass the control to the original program. A program never invokes the VDSO directly but will instead use the wrappers provided by the C standard library (libc). If the ELF auxiliary vector entry for the VDSO is missing, however, then libc will fall back to using the original system call that each virtual system call corresponds to. These original system calls will be intercepted by GHUMVEE's monitor. An alternative solution could be to replace the VDSO with a custom library that leverages GHUMVEE's implicit input replication infrastructure to replicate the master replica's system call results to all slave replicae.

3.2.1 Evaluation and Comparison with other MVEEs

To the best of our knowledge, GHUMVEE is the only existing MVEE that handles the RDTSC instruction correctly. Along with Hosek and Cadar, who independently proposed a solution of their own, we were also the first to handle system calls in the VDSO correctly [53]. Our solutions proposed solutions have a minimal performance impact on the many applications we tested. The RDTSC instruction is typically only used during the startup and shutdown of a program to, e.g., measure the run time of individual threads. Our proposed solution for the VDSO does significantly impact the latency on executing invidual timing-related system calls. In Chapter 6, we propose a new monitor design that reduces this impact to a bare minimum.

[4]GPL will be released under a BSD-style license by Q4 of 2015

3.3 File Operations

Multi-Variant Execution should be transparent to the replicae and to external observers. GHUMVEE therefore uses the master call mechanisms to ensure that I/O operations are only performed once. With this mechanism, only the master replica performs the actual I/O operations, and the monitor will replicate the results to the slave replicae. Intuitively it might make sense to also use master calls for system calls that open, modify or close file descriptors. Regular files might, however, be mapped into the replicae' address spaces using the file mapping API. If we would apply the master call mechanisms to such files, then any subsequent file mapping request would fail in all slave replicae. GHUMVEE therefore allows slave replicae to open, modify and close file descriptors for regular files.

The master call mechanism must still be used to open, modify and close other file descriptors such as sockets, however. Certain system calls, such as sys_accept operate only on file descriptors associated with sockets that are in listening state. Since only one socket can listen on each port, GHUMVEE uses master calls for all socket operations.

Since some file descriptors are opened only in the master replica and some are opened in all replicae, the same file descriptor might have different values in the different replicae. As GHUMVEE must ensure that the multi-variant execution is transparent to the replicae, the monitor replicates the same file descriptor values to all replicae, regardless of whether or not they have actually opened the file. Whenever the replicae perform a normal system call that they must all execute, GHUMVEE will map the replicated file descriptor value back to the original file descriptor value at the system call entrance site. When the same call returns, GHUMVEE will map the original file descriptor value back to the replicated file descriptor value.

3.3.1 Evaluation and Comparison with other MVEEs

Although little details on how other MVEEs handle file descriptors are available, we assume that most of them use a similar solution to ours. One notable exception is Orchestra. Orchestra's monitor performs all I/O operations on behalf of the replicae. Replicae running in Orchestra therefore do not open any file descriptors other than those that correspond to regular files.

3.4 Signal Handling

The many intricacies of the ptrace API, the one-size-fits-all nature of signals in general and the myriad of scenarios in which signals may be delivered make correct handling of asynchronous signals one of the most challenging problems when implementing an MVEE. The sad consequence is that almost no existing MVEEs fully support signal handling. As we showed in Table 2.2, GHUMVEE is, to the best of our knowledge, the only security-oriented MVEE to support, e.g., system call interruption.

3.4.1 Introduction

UNIX systems use signals as a general-purpose mechanism to send notifications to processes [75]. Each such notification has a signal number associated with it and the signal number generally defines the meaning of the notification. When a program performs an invalid memory access for example, the kernel sends a SIGSEGV signal to that program.

Programs have a wide extent of control over how the kernel should treat the signals that are sent to the program. The kernel stores a **blocked signal mask** for each program thread. Every signal number corresponds to one of the bits in this mask. If the bit for a specific signal is set, and that signal is sent to the thread, then the kernel will store the signal's information in a **pending signal queue** and defer further handling of the signal until the signal is unblocked.

The kernel consults the program's **signal handler table** for instructions on how to handle unblocked signals. The signal handler table is a fixed-size array of **signal action rules**, with each rule corresponding to one signal number. Programs can choose whether or not to share the same signal handler table among all their threads. A signal action rule can specify that the corresponding signal should be ignored. The kernel will then discard the signal whenever it attempts to deliver it to the program. Another possibility is to register a signal handler function. The kernel will invoke this function whenever it delivers the associated signal.

We can distinguish between two kinds of signals. **Control-flow signals** such as SIGSEGV are sent as a direct consequence of a program's normal control flow. The program cannot continue executing until the kernel has handled the control-flow signal. If a control-flow signal is not blocked and the program has registered a signal handler function for the signal in the handler table, then the signal will be delivered synchronously. **Asynchronous signals** on the other hand originate from an external source and the program may continue executing while the kernel is handling the delivery of an asynchronous signal.

When the kernel decides to invoke a signal handler function, it first stores a backup of the receiving thread's register context. The kernel then transfers control to the signal handler function, possibly through an intermediate dispatcher function. The signal handler function can then deal with the event that caused the signal to be sent in any way it sees fit. This includes invoking system calls and/or terminating the program. A signal handler function returns by invoking a sys_sigreturn system call. The kernel restores the original register context of the program thread in this system call.

3.4.2 System Call Interruption

The kernel cannot invoke a signal handler function while the receiving thread is executing a blocking system call. Instead, it will interrupt this system call, force the system call to return an error code, and adjust the instruction pointer such that the system call will automatically be restarted when control returns to user space.

Some system calls will be interrupted due to the delivery of a signal, even if the program has not registered a signal handler function for that signal. The system call will be restarted immediately in that case. The process of a restarting a system call is completely transparent to the program itself. If the program is being debugged, however, the debugger may observe the interruption of the system call.

If the program has registered a signal handler function, the kernel will transfer control to said function using the mechanism described above. When this signal handler function returns through an invocation of sys_sigreturn, the kernel will once again restore the original context and transfer control back to the program. Since the kernel has adjusted the instruction pointer prior to invoking the signal handler function, the original system call will now restart.

3.4.3 Signal Delivery under ptrace

The process of delivering a signal to a program being debugged is more complicated. A signal that is sent to a debugged program thread is initially stored in the pending signal queue for that thread. The thread will enter the **signal-delivery-stop** state as soon as the signal is unblocked and as soon as the program leaves any other ptrace-stop state [74]. These other ptrace-stop states include **syscall-stop** and **group-stop**. The debugger observes the transition to **signal-delivery-stop** through the return value of the sys_wait call.

The signal that caused the transition to the stop state stays pending until the thread is resumed by the debugger. At that point, the debugger must decide whether discard or inject the signal. If the debugger decides to inject the signal, the kernel will attempt to deliver it using the mechanism described in Section 3.4.1.

3.4.4 Synchronous Delivery of Asynchronous Signals

Signal handler functions may invoke system calls and/or modify the program state. Security-oriented MVEEs that execute the replicae in lock-step must therefore handle asynchronous signals with great care, so as to avoid introducing inconsistencies in the replicae. To this end, GHUMVEE defers the delivery of asynchronous signals until the replicae reach the next system call RVP. We do this by initially discarding any incoming asynchronous signal in the **signal-delivery-stop** state and by re-sending the signal from the monitor when it can guarantee that the replicae will be in equivalent states upon delivery of the signal. GHUMVEE asserts this equivalence by waiting for the replicae to reach a RVP before sending the signal. One minor problem has to be dealt with, however. Since most system call RVPs correspond to ptrace-stop states, i.e., replicae are generally in the **syscall-stop** state when they reach a system call RVP, the monitor must resume the replicae after sending the signal before they can transition into the **signal-delivery-stop** state. After being resumed, the replicae may have executed an arbitrary number of instructions before they enter the **signal-delivery-stop** state, however. This can once again cause inconsistencies. To prevent this from happening, GHUMVEE injects a tight infinite loop into one of the executable code sections in each replica's address space. This infinite loop does not modify the program state, nor does it invoke any system calls. Prior to sending a signal, GHUMVEE ensures that the

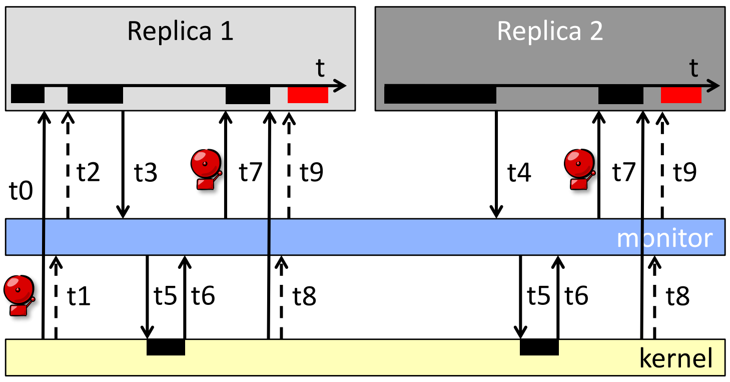


Figure 3.1: Synchronous Signal Delivery in an MVEE

replicae are stopped at a system call exit. Then, GHUMVEE stores a back-up of each replica's register context and overrides the instruction pointer to point to this small infinite loop. At this point, GHUMVEE can safely send the signal.

Figure 3.1 illustrates the scenario where an asynchronous signal is sent to the master replica. This is the most common scenario. Due to GHUMVEE's master call mechanism, the slave replicae typically are invisible to the rest of the system. Most signals are therefore only sent to the master replica. One notable exception is the SIGCHLD signal, which the kernel will automatically send to a parent process whenever one of its child processes dies. Handling the SIGCHLD signal is a trivial extension of the above scenario. In the figure, at $t0$, we see that the kernel stops the master replica because a signal is sent to it. The monitor receives the corresponding notification at $t1$ and resumes the master at $t2$ while discarding the signal. The replicae then reach a RVP at $t4$, after the master had already reached the **syscall-stop** state at $t3$. At this point, the monitor cannot send the signal since both replicae are stopped. Instead, it will resume them and wait until the system call returns. GHUMVEE will store the back-up of the register context *before* the system call and override the system call number with that of sys_getpid. When this sys_getpid call returns at $t5$-$t6$, GHUMVEE will override the instruction pointer so that the replicae will jump to the infinite loop. GHUMVEE then sends the original signal at $t7$ and resumes the replicae. At $t8$, the replicae will have entered the **signal-delivery-stop** state and GHUMVEE will have been notified. GHUMVEE then resumes the replicae while injecting the signal at $t9$. The replicae then enter the signal handling function associated with the signal.

3.4.5 System Call Interruption in GHUMVEE

Delivering a signal while all replicae are executing user-space code, as illustrated in Figure 3.1, is straightforward. Things quickly get more complicated, however, when the replicae are executing a blocking system call. As explained in Section 3.4.2, blocking system calls are interrupted by the delivery of a signal and will restart automatically, possibly after the calling thread has invoked its signal handler for the signal that had interrupted the system call. While this process is transparent to the replica itself, it is not transparent to GHUMVEE.

Whenever the kernel interrupts a blocking system call executed by one of the replicae, GHUMVEE will first see that system call returning with an error code as the return value of that system call. The exact error code indicates how the kernel would normally have handled the further processing of the signal, had GHUMVEE not been attached to the replica. GHUMVEE cannot see the signal that had interrupted the replica until it is no longer in **syscall-stop** state. GHUMVEE therefore cannot know whether the interrupting signal currently has an associated signal handler and whether or not the signal is currently blocked or ignored. GHUMVEE therefore always restarts the interrupted system call handler before attempting to deliver any signal.

The kernel informs GHUMVEE about the signal after it has restarted the replica, but before the restarted replica is back at the system call entrance site. GHUMVEE can therefore decide whether or not to take further action at the entrance site of the restarted system call. It must, however, still ensure that the delivery of the signal happens synchronously. For this, all replicae must be synchronized at the same RVP before the injection of the signal is attempted. This is anything but a given because there are two common situations in which not all replicae get interrupted by the delivery of the signal.

First, the signal may have been delivered during a **master call**. Whenever the replicae invoke a system call that GHUMVEE dispatches as a master call, only the master replica will invoke the original system call, while the slave replicae will invoke a sys_getpid call.

If the master replica gets interrupted by a signal, then GHUMVEE must only restart the system call for the master replica. Then, when the master replica is back at the entrance site for this master call, GHUMVEE can decide whether or not the signal should be injected. If the signal is to be injected, GHUMVEE will replace the system call number in the master replica by that of sys_getpid and resume only that replica. The signal can then be safely injected at the return site of this sys_getpid call, since the slave replicae will also be stopped at the same return site.

Second, the signal may also have been delivered during a **normal call**, but only some of the replicae may have been interrupted. This is a scenario that frequently occurs because certain blocking system calls return immediately if a signal is pending. If a signal is sent to all replicae at the same time but some replicae begin executing the system call before the signal is sent and some begin executing the system call after the signal is sent then the first set of replicae will be interrupted by the signal, while in the other replicae, the system call will simply return.

GHUMVEE will restart the current system calls in all replicae in this scenario. For the replicae that did not have their system calls interrupted by the signal, however, GHUMVEE must manually adjust the instruction pointer and restore the original register context before it can restart the system call.

3.4.6 Evaluation and Comparison with other MVEEs

GHUMVEE's mechanism for handling asynchronous signal delivery is optimized for correctness, rather than performance. During our evaluation, we have indeed concluded that almost every program that relies on signal handling still functions correctly inside GHUMVEE. The only exception we have found is the john-the-ripper program in the phoronix 4.8.3 benchmark suite. This program waits for signals to be delivered in a busy loop, in which no system calls are used. Therefore, if GHUMVEE intercepts a signal that is delivered to the replicae, it will indefinitely defer the delivery of the signal because the replicae never reach another system call RVP. One solution could be to start a timer when a signal is intercepted and to force the delivery of the signal when the timer expires.

Orchestra's mechanism for signal handling is optimized for performance, rather than correctness [114]. Orchestra uses a heuristic to determine whether a signal can be safely delivered, even if its replicae have not reached a system call RVP yet. Orchestra does however not handle signals that interrupt system calls correctly.

VARAN's signal handling mechanism is ideal with respect to performance and correctness [53]. VARAN is an In-Process monitor and, therefore, does not rely on the ptrace API. Furthermore, VARAN forces its follower replicae to not invoke system calls at all. Instead, these follower replicae just wait for the results of the leader replicae's system calls. In VARAN, the leader replica accepts and processes incoming signals without delay. While the follower replicae generally do not receive signals at all, the leader replica will log the meta-data associated with the signal into the event streaming buffer. This meta-data provides the follower replicae with sufficient information to replay the invocation of the signal handler truthfully.

3.5 Address-Sensitive Behavior

Most of the sources of inconsistencies in the behavior of single-threaded replicae can be eliminated or mitigated by the monitor itself. The one notable exception is **address sensitivity**, a problem we have frequently encountered in real-world software. The observable behavior of address-sensitive programs depends on their address space layout. Any form of code, data, or address space layout diversification we use in the replicae can therefore lead to false positive detections by the monitor.

We have identified three problematic idioms that lead to address sensitivity:

- **Address-sensitive data structures.** We have frequently encountered programs that use data structures whose run-time layout depends on numerical pointer values. This practice is especially common among programs that rely on glib, the base library of the GNOME desktop suite. glib exposes interfaces that C programs may use to create, manage and access hash tables and binary trees. The default behavior of these glib data structures is to insert new elements based on their location in memory. When an object is to be inserted in a glib hash table, glib will calculate a hash value based on the object's location to determine which bucket the object should be linked into. glib's binary tree implementation on the other hand will sort its nodes based on their numerical pointer values.

  Applying diversification techniques that modify the address space layout in multiple replicae of a program that contains address-sensitive data structures will yield differences in the replicae' system call and synchronization behavior. In address-sensitive hash tables for example, an insertion of the same logical object in several replicae could trigger a hash table collision and a subsequent memory allocation request (e.g., through a sys_mmap call) in some replicae but not in others.

While it might seem sensible to tolerate small variations in the system call behavior, we typically cannot allow variations in the memory allocation behavior of the replicae, which we are bound to see in programs with address-sensitive data structures. Variations in the memory allocation behavior cause a **ripple effect** in multi-threaded replicae because tolerating a minor discrepancy early on leads to bigger and bigger discrepancies in the synchronization behavior and, consequently, in the system call of the replicae, to a point where we can no longer distinguish between benign discrepancies and compromised replicae.

Dynamic memory allocators are the instigator of this ripple effect. GNU libc's ptmalloc, for example, attempts to satisfy any memory allocation request by reserving memory in one of its arenas. All accesses to the allocator's internal bookkeeping structures must be thread-safe. It therefore relies on thread synchronization to ensure safety. As we will discuss at length in Chapter 4, GHUMVEE replicates the master replica's synchronization operations in the slave replicae. Thus, if the replicae behave differently with respect to memory allocations, the replicated synchronization information might be misinterpreted by other replicae because it does not match their actual behavior. From that point onwards, such replicae will no longer replay synchronization operations in the same order as the master and will therefore typically diverge from the master with respect to the system call behavior.

- **Allocation of aligned memory regions.** An additional problem we have identified in ptmalloc is its requirement that any memory region it allocates through sys_mmap is aligned to a large boundary of, e.g., 1MiB in Figure 3.2. The operating system only guarantees that newly allocated memory regions are aligned to a boundary equal to the size of a physical memory page. As shown in Figure 3.2, ptmalloc therefore always allocates twice the memory it needs and subsequently deallocates the region before and the

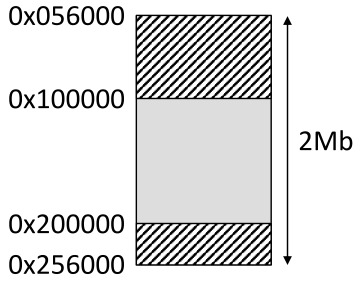


Figure 3.2: Aligned allocation in ptmalloc

region after the boundary. When running multiple replicae that use this memory allocator, the sizes of the deallocated upper and lower regions might differ. Worse yet, in some cases the newly allocated memory might already be aligned to the desired boundary and ptmalloc will therefore only deallocate the upper region. This might trigger false positive detections in MVEEs that execute their replicae in lock-step since some replicae may deallocate the lower and the upper region, while others only deallocate the upper region.

- **Writing output that contains pointers.** Some programs output numerical pointer values. Unlike the previous problematic idioms, writing out pointers often only leads to minor differences in the system call behavior and we have not encountered any cases where writing out pointers triggers a ripple effect. It is therefore sensible to tolerate minor differences in the program output.

  One problem to deal with, however, is that pointers are not always easily recognizable in a program's output. Some programs encode pointers, e.g., by storing them as an offset relative to a global variable or object. Encoded pointers are often smaller than the size of a memory word.

  Similarly, many programs and libraries use partially uninitialized structures as arguments for a system call. The uninitialized portions of these structures may contain leftovers of previous allocations. These leftovers often include pointers. While it can often be considered a bug to pass uninitialized structures to the kernel, there are cases where the programmer is not to blame. An optimizing compiler aligns members of a data structure such as the one in Figure 3.3 to their natural boundary. In this figure, the

```
struct padded_struct {
    char ch1;    // 1 byte
                 // 3 padding bytes
    int i1;      // 4 bytes on 64bits system
    int i2;      // 4 bytes
};
```

Figure 3.3: Padding in data structures

three bytes after variable `ch1` will therefore not be used. These three bytes might therefore never be initialized and might overlap with remainders of previously allocated objects. This can once again lead to minor variations in the output behavior.

All of the above idioms lead to discrepancies in the replicae' system call behavior and/or synchronization behavior. Small variations in the system call behavior can in some cases be tolerated, especially if the variations are limited to the arguments of a single system call. Variations in the synchronization behavior cannot be tolerated, however, as we will argue in Chapter 4.

3.5.1 Diversity Rendez-Vous Points

In order to maintain equivalent system call and synchronization behavior, even if address layout diversification techniques are used, we introduce **implicit input replication agents** and **diversity rendez-vous points (DRVPs)**. We add these DRVPs to the points in the replicae's code where the addresses of objects in address-sensitive data structures are used in pointer arithmetic or branch conditions.

At every DRVP, we insert a call to an implicit input replication agent. When called in the context of the master replica, the replication agent will store the address of the object in a circular buffer that is visible to all replicae, as shown in Figure 3.4. When called in the context of the slave replicae, the agent reads the stored address from the buffer and returns it to the replica so that the recorded address may be used in arithmetic operations and branch conditions.

We developed three components that aid in the implementation of DRVPs and implicit input replication agents:

- **The implicit input replication API:** The implicit input replication API can be used to generate the replication agents and the

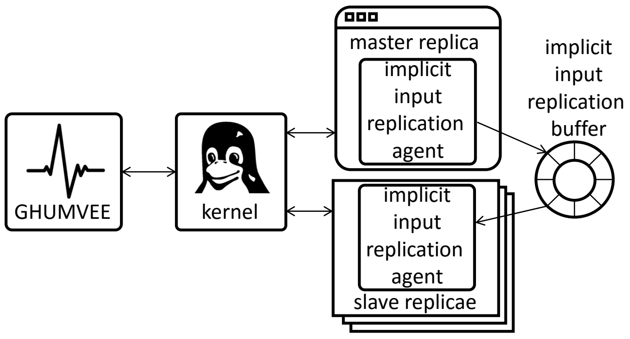

master replica
implicit input replication agent
implicit input replication buffer
GHUMVEE
kernel
implicit input replication agent
slave replicae

Figure 3.4: Using implicit input replication agents to tolerate address-sensitive behavior.

DRVPs. The API consists of a set of preprocessor macros that expand into C functions. These C functions implement the recording and forwarding logic of the replication agent. In the master replica, the generated function can retrieve the implicit input by calling a programmer-specified function and can then record the input into a circular buffer. In the slave replicae, the generated function retrieves the input from the circular buffer.

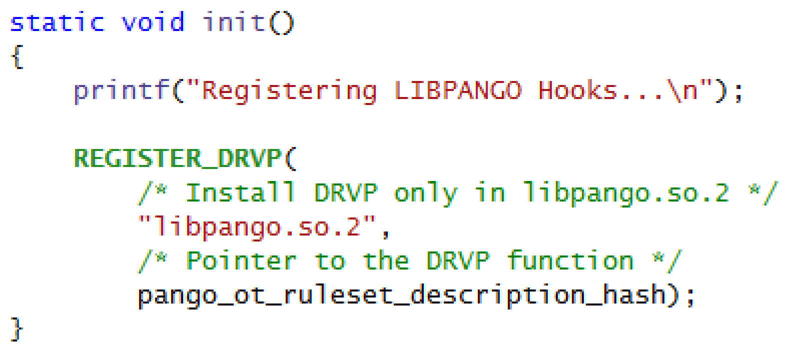
An example of the replication API is shown in Figure 3.5. In this example, we generate a DRVP function for use in the popular pango rendering library. This library stores several types of objects into address-sensitive data structures. In this example, we generate a DRVP for the hash table that stores PangoOTRulesetDescription objects. The intent of this DRVP is to replace the original address-sensitive hash function, pango_ot_ruleset_description_hash. The DRVP function therefore accepts the same arguments as this hash function. In the master replica, the generated function will call the original hash function and it will record the result into the MVEE_PANGO_HASH_BUFFER.

In the slave replicae, the generated DRVP will read the recorded hash from that same buffer. The replication API further allows the programmer to specify whether or not the slaves should also invoke the original function, e.g., if the original function has side effects which may affect future system call and synchronization behavior. It also supports debugging features that are not shown in the figure.

- **The lazy hooker:** The generated DRVP functions can be embed-

```
GENERATE_DRVP_FUNC(guint,
    pango_ot_ruleset_description_hash,
    (const PangoOTRulesetDescription* desc))
{
    FORWARD_INPUT(
        /* return type for the original function */
        guint,
        /* pointer to original function */
        pango_ot_ruleset_description_hash,
        /* arguments to original function */
        (desc),
        /* identifier for the circular buffer */
        MVEE_PANGO_HASH_BUFFER,
        /* should slaves call the original function? */
        SLAVES_DONT_CALL_ORIGINAL_FUNCTION,
        /* should slaves check if their result matches
        the master's result? */
        SLAVES_DONT_CHECK_RESULT,
        /* execute the original function before locking
        the buffer and loggging the result */
        EXECUTE_BEFORE_LOCK);
    return result;
}
```

Figure 3.5: Usage example for the implicit input replication API.



```
static void init()
{
    printf("Registering LIBPANGO Hooks...\n");

    REGISTER_DRVP(
        /* Install DRVP only in libpango.so.2 */
        "libpango.so.2",
        /* Pointer to the DRVP function */
        pango_ot_ruleset_description_hash);
}
```

Figure 3.6: Registering a DRVP function with the lazy hooker.

ded in the replica by registering them with a shared library, which we call "the lazy hooker". This library monitors the dynamic loading process of the replicae and can determine whether or not a DRVP should be installed. Figure 3.6 shows how to register the DRVP function we generated in Figure 3.5. Using the REGISTER_DRVP macro, we indicate that the lazy hooker may only insert the specified DRVP function in the libpango.so.2 library, and that the call to the DRVP function must be inserted in the function with the same name as the DRVP function. Alternatively, by specifying "*" as the library name, the programmer can request that the DRVP be inserted in each library that exposes a function with the same name as the DRVP function.

At the time of the registration, the lazy hooker may insert the DRVP function immediately, if the specified library has already been loaded, or it can defer the insertion until the program loads the library.

- **The LinuxDetours library:** We insert the DRVP functions using LinuxDetours, a run-time code patching library we developed for use in GHUMVEE. The library is named after Microsoft's Detours library and implements a subset of the official Detours API [56]. LinuxDetours can redirect calls to functions and generate trampolines that may be used to call the original function, without interception. In our example, we redirect all calls to the pango_ot_ruleset_description_hash function to our DRVP function. Our DRVP function then uses the generated trampoline to invoke the original function in the master replica.

The method described above works especially well to support replicae with address-sensitive data structures, such as some hash tables. Address-sensitive hash tables require a dedicated hash function. This dedicated hash function typically does nothing more than performing a simple, in this case address-sensitive, hash calculation. The function is also typically neutral with respect to the system call behavior of the replicae and usually does not affect the replicae's synchronization behavior. Using the aforementioned method, it is straightforward to create dedicated DRVP functions that can replace the original hash functions.

This method does not work well for memory allocators with special alignment requirements. For example, ptmalloc does not have a function that is dedicated to requesting new memory regions from the kernel (through sys_brk or sys_mmap). All allocation requests are inlined in larger functions. Furthermore, the address-sensitive functionality of ptmalloc does have side effects (i.e., a sys_mmap call), so we cannot blindly replicate the results of the address-sensitive functionality in the slave replicae. Instead, we chose to introduce a DRVP directly into ptmalloc's source code. Right after ptmalloc allocates a new block of memory through sys_mmap, we added a sys_all_heaps_aligned system call. The latter system call is non-existing but GHUMVEE intercepts it and interprets it as a real system call nonetheless. The sys_all_heaps_aligned system call returns 1 if the most recently allocated memory regions were all aligned to ptmalloc's desired boundary or 0 otherwise. If the call returns 0, the DRVP function unmaps the most recently allocated heap in all replicae and forces them to fall back to unaligned allocation.

3.5.2 Evaluation and Comparison with other MVEEs

We applied the implicit input replication API to run two popular, though now outdated, desktop programs: the Firefox 3.6 browser and the LibreOffice 4.5 office suite. We had to embed five replication agents in total in order to run these programs reliably with Address Space Layout Randomization enabled. To evaluate the developer effort required to write these replication agents, we calculated their sizes in lines of C code. The results are shown in Table 3.1.

To the best of our knowledge, GHUMVEE is the only MVEE to date that has any provisions to eliminate inconsistencies resulting from address-sensitive behavior.

infrastructure	implicit input replication API	lazy hooker	LinuxDetours	total
lines of C code	260	355	180	795

agents/DRVPs	glib	gtk	orbit	pango	libreoffice	total
lines of C code	105	54	78	54	183	474

Table 3.1: Developer effort for the implicit input replication agents.

3.6 Conclusions

Older MVEEs build on the assumption that all program input either originates from the system call interface, or can be stopped at the system call interface by, e.g., disallowing the use of shared memory. In this chapter, we described several cases in which this assumption is false. We presented an overview of all inconsistencies one typically encounters when replicating single-threaded replicae and suggested solutions for all inconsistencies that cannot be eliminated by replicating input at the system call interface.

Chapter 4

Replication of Multi-Threaded Programs

In the previous chapter, we presented *implicit input replication agents* to deal with implicit inputs that introduce system call consistency violations, such as randomized virtual code and data addresses that effect the system call behavior through address-dependent computations [133]. These agents log the implicit inputs in one replica, the so-called master, and replicate those inputs in the other replicae, the so-called slaves, thus restoring system call consistency. Together with the system calls, the points at which these agents intervene form the so-called *rendez-vous points* RVPs of the replicae.

In this chapter, we focus on a big weakness of existing secure MVEEs with respect to system call consistency: secure replication of non-deterministic multi-threaded applications. In real-life multi-threaded programs, even the security-sensitive system calls that should be monitored most strictly often differ between replicae as a result of their non-deterministic scheduling. Because of the lock-step system call execution and monitoring requirement, security-oriented MVEEs cannot tolerate those divergences.

Two broad classes of techniques could potentially alleviate this problem. First, a Deterministic Multi-Threading (DMT) system can be embedded in the program to enforce a fixed thread schedule in all replicae [10, 15, 17, 37, 40, 81, 82, 89, 97, 108, 145]. In the context of an MVEE, however, all existing DMT systems fall short. To establish a deterministic schedule, they all rely, in one way or another, on a token. Some systems only allow threads to pass this token when they invoke a synchronization operation. This approach is incompatible with threads that deliberately wait in an infinite loop for an event such as the delivery of a signal to trigger because such threads may never invoke a synchronization operation. Other systems allow threads to pass their token when they have executed a certain number of instructions. Such systems cannot tolerate variations in the program execution and are therefore incompatible with most code diversification techniques as well as the implicit-input replication agents.

Alternatively, we can accept non-determinism and require only that all replicae execute in the same non-deterministic order. Online Record/Replay (R+R) systems can provide this guarantee by logging the execution in one replica and replaying it in the other replicae [11, 12, 69]. R+R systems are less sensitive to variations in the program execution, which we typically see with diversified replicae. But in order to use them in a security-oriented MVEE, they need to be adapted to become address-agnostic and to support programs that use ad hoc (i.e. non-standardized) synchronization primitives or lock-free algorithms. Furthermore, for embedding an R+R system in a security-oriented MVEE we need to ensure that any new functionality introduced in the replicae must be neutral with respect to the RVPs, and we need to secure the communication channel that is used to convey the information about the recorded execution from the master replica to the slave replicae.

This chapter makes four contributions. First, we present four R+R-based synchronization replication agents that record synchronization operations in a master replica and replay them in the slaves, thus ensuring system call consistency. The agents are address-agnostic and system-call-neutral, and hence compatible with existing secure MVEEs and implicit-input replication agents. The most efficient agent communicates over a channel that is secured against malicious communication by attackers.

Second, we present a practical strategy to extend our R+R-based systems to support ad hoc and lock-free synchronization, which we typically see in many low-level libraries.

Third, we report how we integrated our replication agents into GNU's *glibc* and how we applied the aforementioned strategy to four commonly used system libraries: GNU's *libpthreads*, *libstdc++* and *libgomp*. This integration enables support for data race free C and C++ programs that use the *pthread* and/or *OpenMP* programming models.

Finally, we extensively evaluate the run-time performance of our

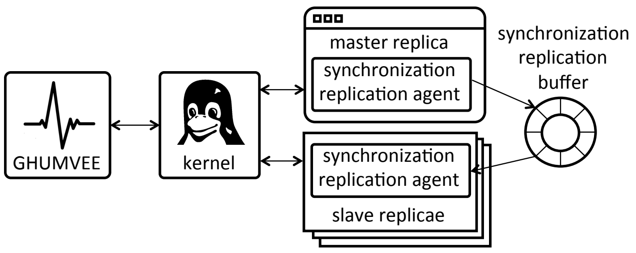

master replica
synchronization replication agent
synchronization replication buffer
GHUMVEE
kernel
synchronization replication agent
slave replicae

Figure 4.1: System overview

replication agents, the implementation effort that went into their integration into the aforementioned libraries, and the security of the proposed features.

4.1 Replication of Multi-Threaded Programs

To monitor the replicae, GHUMVEE uses the ptrace and process_vm_* Linux APIs, which we discussed in depth in Chapter 2. As the use of these APIs involves context switching, they introduce significant latencies in the interaction between the monitor and the replicae. This makes them unacceptable for replicating synchronization events, which occur frequently in many programs and which are often handled entirely in user space in the original programs to optimize performance. For example, we observed gcalctool, a simple calculator from the GNOME desktop environment performing 1.8M futex operations during its 400 ms initialization, almost all of which were uncontended and hence handled in user space. Interposing all those operations with system calls and ptrace made the initialization time grow to over 370 seconds, a slowdown with a factor 925!

To avoid such an unacceptable overhead, our alternative solution consists of a synchronization replication agent that replicates all synchronization events entirely in user space.

4.1.1 Synchronization Replication Agent

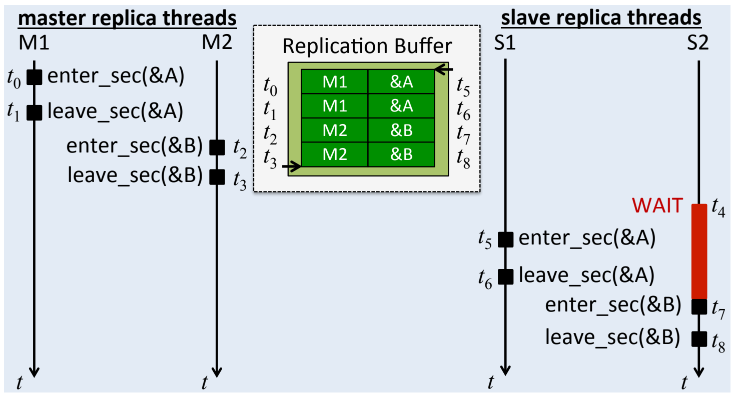
We enforce an equivalent execution in all replicae by injecting a synchronization replication agent into their address space, as shown in Figure 4.1. At run time this agent forces the master replica to capture the order of all inter-thread synchronization operations, hereafter referred to as *sync ops*. The agent logs the captured order in a circular, shared buffer that is visible to all replicae. This buffer is mapped with read-/write permission in the master replica and with read-only permission, and at different addresses, in the slave replicae. In the slave replicae, the agent uses the captured order to enforce an equivalent replay of sync ops.

To capture the sync op execution order, we wrap them in a small critical section at the source code level. Within the critical section we first log information about the sync op in the first available slot of the buffer and then perform the original op. Depending on the agent we use, the information about the sync op consists of the thread ID, the memory word that was affected by the op, and the op's type.

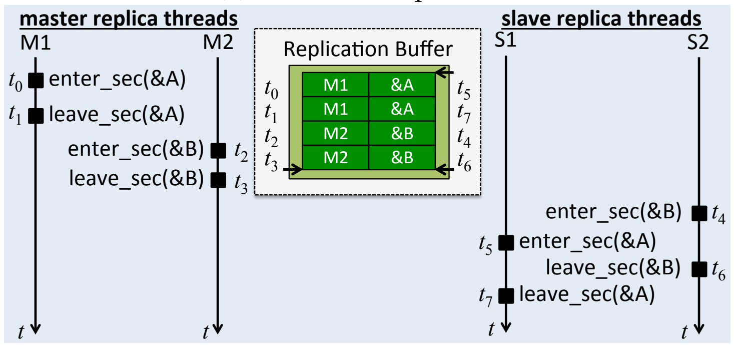
The replication agent must be available to the entire program, including any loaded shared libraries. So we chose to implement the agent in glibc, at the lowest possible level in the user-mode portion of the software stack where it is exposed to the program itself and to all shared libraries.

The same agent is used in the master and slave replicae: Identical instances of glibc are loaded into the master and slave replicae when they are launched, though they might be loaded different addresses in each replica. When an instance is later invoked at run time, it needs to know whether it is invoked in a master or slave replicae, to either capture or replay the sync op order. We therefore dynamically initialize the agent in each replica. Soon after a replica is launched by the MVEE monitor, its agent invokes a system call that is intercepted by the monitor. Through this system call, the agent passes the location of its status flags to the monitor. At that point, and at each later intervention from the monitor in the replicae, the monitor can configure the agent instance as a master or slave agent. Through the status flags, the monitor can also disable the agent when the replicae are executing a single thread, and enable the agent when the replicae (are about to) start executing multiple threads. From that initial configuration onwards, the agent in each replica communicates only with the agents in the other replicae. With the exception of being enabled/disabled, the agents do not communicate with the monitor. This avoids the extensive context switches that would result from using the ptrace or process_vm_* APIs for replication.

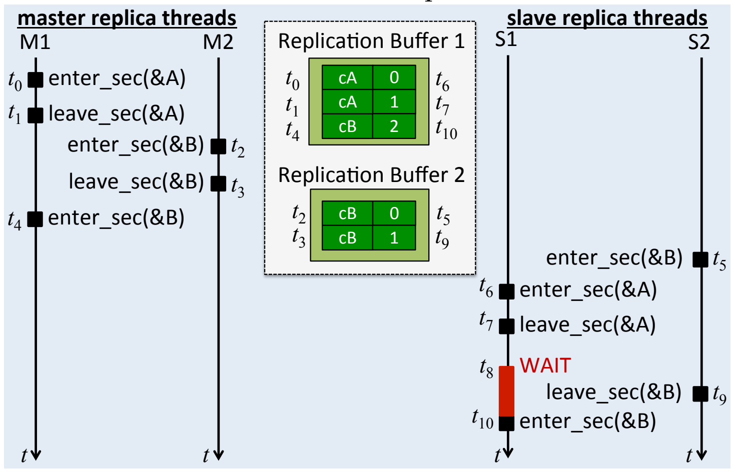
While the high-level principles of our replication agents are reminiscent of online R+R techniques, we cannot trivially adopt them. Within



(a) Total-order replication



(b) Partial-order replication



(c) Wall-of-clocks replication

Figure 4.2: Replay sequences with three replication strategies

an MVEE, it is critical that any new functionality injected into the original code is *neutral with respect to RVPs*. Specifically, we have to ensure that if the new functionality introduces a new RVP, this RVP is introduced in all replicae at the exact same point in the program execution and with equivalent arguments. Furthermore, because a secure MVEE needs to enforce lock-step execution on the replicae, we need to replicate information actively and with minimal delay to avoid that slave replicae delay the master replica too much. We therefore implement replication agents that make the recorded information *visible immediately*, rather than broadcasting it periodically.

These two design decisions have far-reaching consequences. First, the RVP-neutrality constraint prevents us from using dynamic memory allocators, because those use sync ops to coordinate multi-threaded access to the memory, and introduce system call RVPs to allocate additional memory pages and to change protection flags.

Second, since we want any information to be visible immediately in the slave replicae, the agent cannot perform any post-processing on the recorded information. This prevents us from compressing the recorded information to reduce our agents' memory bandwidth requirements.

Third, our agent must support *diversified replicae*. It can therefore not assume that the master and slave replicae are fully identical. For example, the same mutex might be found at different addresses in the different replicae. Consequently, the recording side of the replication agent must record its information in a manner that is *address-agnostic*.

4.1.2 Replication Strategies

To replay the sync ops in the slave replicae, several approaches are available that trade CPU cycles off against memory pressure. We have implemented three replication strategies that meet the aforementioned constraints.

Total-order replication agent

Our total-order (TO) replication agent replays all sync ops in the exact same order in which they happened in the master replica. Figure 4.2(a) shows two threads that execute under GHUMVEE's control. In the master replica, thread `M1` first enters and leaves a critical section protected by lock `A` at times t_0 and t_1 resp. At those times, the wrappers of the corresponding sync ops log the activities of thread M1 in the replication buffer. Next, thread M2 in the master replica enters and leaves a critical section protected by lock B at times t_2 and t_3 resp. These events are also logged in order in the buffer. Right after t_3, the buffer holds the contents indicated in the figure. Time stamps to the left and right of the buffer mark the time the buffer elements are produced and consumed resp. The arrows on the left and right denote the position of the producer and the consumer pointers resp. right after t_3.

In the slave replica thread S2, corresponding to M2 in the master replica, reaches the critical section protected by lock B first, at time t_4. At that time, the first element in the buffer indicates that synchronization events in the master replica occurred in thread M1 first, so thread S2 is stalled in the wrapper of the sync op in enter_sec. Only after the first two elements in the buffer are consumed in thread S1 at times t_5 and t_6, can thread S2 continue executing. Thus, even though the two critical sections protected by locks A and B are unrelated, thread S2 is forced to stall until thread S1 has replayed the operations performed by thread M1.

This agent is trivial to implement, but not very efficient: The lack of lookahead by consumers introduces unnecessary stalls as indicated by the red bar in Figure 4.2(a).

Partial-order replication agent

Our partial-order (PO) replication agent is more efficient. It only enforces a total order on dependent sync ops. This agent may replay independent sync ops in any order, as long as it preserves sequential consistency within the thread. The PO agent is more complex and introduces more memory pressure because the agents in the slave threads have to scan a window in the buffer to look ahead. However, it typically introduces much less stalling and generally outperforms the TO agent. In Figure 4.2(b), we see the exact same order of events as in Figure 4.2(a) until t_4. This time, however, thread S2 may enter the critical section without delay at t_4 because the enter_sec operation does not depend on either of the operations that preceded it in the recorded total order.

Conceptually, there are significant similarities between this agent and online R+R techniques such as LSA [11] and offline R+R techniques such as RecPlay [110]. However, our agent captures events at a finer granularity of *sync ops* instead of *pthread*-based synchronization operations. Furthermore, our agent is fully RVP-neutral. These are relatively minor differences, however, as techniques like LSA can be adapted to capture at a lower granularity and to use only statically allocated memory. A more fundamental difference is that our agent also supports diversified replicae. With queue projection, LSA discards the per-thread order of synchronization operations and only maintains the per-variable order. With diversified replicae, the same logical variable might be stored at different addresses in different replicae. Our agent therefore relies on the per-thread order to determine which logical variable is affected by each synchronization operation.

Although the PO agent eliminates unnecessary stalling, it still suffers from poor scalability. The master replica must safely coordinate access to the circular buffer by determining the next free position in which it can log an operation. If many threads simultaneously log synchronization events, this inevitably leads to read-write sharing on the variable that stores the next free position. A similar problem exists on the slave replicae's side because they must keep track of which data has been consumed. With multiple slave replicae, this also leads to high sharing and, consequently, high cache pressure and cache coherency traffic.

Wall-of-clocks replication agent

The above observation led us to the design a third agent. This wall-of-clocks (WoC) agent assigns each distinct memory location that is ever involved in a sync op to a logical clock. These clocks capture "happens-before" relationships between related sync ops [67]. Similar to, e.g., plausible clocks, but without using clock vectors, our clocks only capture the necessary relationships [127].

In Figure 4.2(c), lock `A` stored at address `&A` is assigned to clock `cA`. Lock `B` is similarly assigned to clock `cB`.

On the master side, the agent logs the identifier of the logical clock associated with each sync op, as well as that clock's time. After logging each sync op, the agent increments the logical clock time of the associated clock.

In this agent, the logging is no longer done in a single circular buffer. Instead there is one circular buffer per master thread, such that each buffer has only one producer. In Figure 4.2(c), master thread `M1` only communicates with slave thread S1 through buffer 1, whereas thread M2 only communicates with thread S2 through buffer 2. This design avoids the contention for access to the shared buffers.

Neither the master nor the slave replicae need to propagate their current buffer positions to other threads. Furthermore, the master's logical clocks do not need to be visible to the slaves. The information contained within the circular buffers suffices for the slave replicae to replay the same clock increments on their own local copies of each clock.

In Figure 4.2(c), thread M1 first enters a critical section protected by lock A at time t_0. The agent observes that the current time on logical clock cA is 0. It records the clock and its time in buffer 1 and increments the clock's time to 1. At time t_1, the agent logs the exit from the critical section in buffer 1. This time around, the logical clock time is 1.

A similar situation then unfolds in thread M2 at time t_2. This time though, the critical section is protected by lock B, of which the associated memory location is assigned to clock cB, whose initial time also is 0. This information is logged in circular buffer 2, along with information regarding the exit of the critical section in thread M2 at time t_3. At that point, clock cB is incremented to 2.

In thread M1, a third critical section is entered at time t_4, which is again protected by lock B. This event involving logical clock cB is logged in buffer 1 with clock time 2.

On the slave replica's side, the threads are scheduled differently in our example. There, thread S2 reaches a sync op first, at time t_5. The agent observes in buffer 2 that it must wait until clock cB reaches time 0. Since this is the initial time on the slave's copy of that clock, the operation can be executed right away and thread S2 will increment the time on its copy of cB to 1. If we suppose that thread S2 is then preempted and thread S1 gets scheduled, S1 will enter and leave the critical section protected by lock A at times t_6 and t_7, consuming the first two entries in buffer 1, thereby incrementing the slave copy of clock cA to 2.

The third operation in thread S1 at time t_8 is the most interesting. In the first replication buffer, the slave agent observes that the sync op to enter a critical section has to wait until its associated logical clock cB has reached time 2. However, in the slave, that clock's time was last incremented at time t_5, i.e., to the value of 1. Thread S1 must therefore wait until some other slave thread has incremented the time on cB. This will happen at time t_9 in thread S2. Shortly thereafter, the agent code executing in thread S1 will observe that cB has reached the necessary value, and at t_{10} S1 will enter its second critical section.

With this WoC, the replication agent only inserts accesses to shared data, and hence coherence traffic, for two reasons. First, it introduces accesses to replication buffers shared between corresponding threads in the master and slave replicae. This is a fundamentally unavoidable form of overhead required to replicate the synchronization behavior from the master to the slave replicae.

Secondly, the agent inserts accesses to shared clocks whenever multiple threads in the original program were already contending for locks at shared memory locations. While these extra shared accesses in the replication agents still introduce some overhead, we do expect the overhead to scale with the pre-existing resource contention in the original application. In other words, if the original application uses contended global locks that decrease the available parallelism, the replication agent will hurt it further. However, if the original application involves a lot of synchronization, but that synchronization is performed using uncontended local locks, the WoC replication agent will not introduce contended traffic within the master or slave replicae either.

As we will see in Section 4.2, the WoC agent consistently outperforms the other agents on almost every benchmark. Most importantly, as is the case with plausible clocks in general, the replication will always be correct [127].

One important remark remains to be made, however. While the WoC agent is certainly the more elegant and more efficient of the three proposed designs, it is not fully optimal. Due to the RVP-neutrality constraint, we cannot dynamically assign each memory location to its own private clock. Instead, we have to pre-allocate a fixed number of clocks statically and we have to assign lock memory locations to one of those clocks based on a hash of their memory address. Because we want to use a cheap hash function, hash collusions are quite likely. Any such collusion results in an m-to-1 mapping between multiple locks and each clock. In other words, the WoC agent is bound to assign some non-conflicting memory locations to the same logical clock. When this happens, this introduces unnecessary serialization and hence potentially also unnecessary stalls in the slave replicae.

Our WoC agent is similar to Respec [69], although it does not share any part of its implementation. It differs from other clock-based techniques, however, in that it does not use thread clocks. Instead, our agent relies solely on the logical clock it assigns to each memory lo-

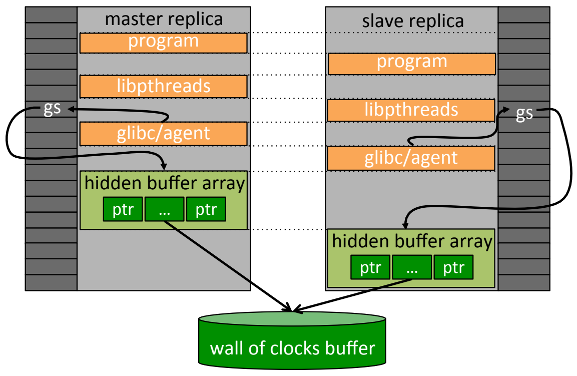


Figure 4.3: Hidden buffer array access to the replication buffer.

cation. In the ideal case, our agent therefore only needs to read and update the value of one clock to replay a synchronization operation. Techniques that rely on Lamport clocks (e.g. ROLT [70]) by contrast need to read and update the values of two clocks: the local thread clock and the synchronization variable's associated clock. Techniques that rely on vector clocks (e.g. RecPlay [110]) need to read the value of at least $n + 1$ clocks (with n the number of threads in the program): the local thread clock, the synchronization variable's clock, and the thread clocks of all other threads. The reason why our agent does not need local thread clocks is that it records into a per-thread buffer, rather than a globally shared buffer. Therefore, a thread clock would never have to be synchronized with other thread clocks, which eliminates the need for such a clock altogether. Furthermore, the fact that our agent assigns each memory location to a statically allocated clock implies that the agent can be applied transparently and that it respects RVP-neutrality.

4.1.3 Secured wall-of-clocks agent

The agents implementing the three replication strategies as discussed in the last sections are not very secure: They forward information through a circular buffer that is shared among all replicae. This buffer easy to locate since all three of these agents store a pointer to it in a thread-local variable. Despite of the code reuse countermeasures we have in place [130], attackers could exploit the fact that an easily locatable communication channel between the replicae exists to set up an attack that can compromise multiple replicae.

As the WoC agent outperforms the other two agents on average, we build on that agent to present an alternative, secured design that relies on the *hidden buffer array* (HBA) shown in Figure 4.3. This page-sized array stores pointers to hidden buffers. Upon startup, a replica can request that this HBA be allocated by GHUMVEE and subsequently map it into its own address space using the System V IPC API [77]. GHUMVEE intercepts and manipulates this mapping call such that the pointer to the HBA is not returned to the program. At the same time though, GHUMVEE overrides the base address of the replica's gs segment so that it points to the HBA.

The reason to override this address is that the x86_64 architecture supports addressing of 48-bit (or bigger) pointers and has therefore disabled most of the original x86 segmentation functionality. The gs and fs segment registers may still be used as additional base registers, however, and by consequence all gs or fs-relative memory accesses are still valid. It is extremely uncommon to still find such accesses in x86_64 software, however. Furthermore, x86 processors do not allow user-space instructions to read the segment registers. The gs and fs segments can therefore be used to store pointers that are hidden from the user-space software.

At a fixed offset within the HBA we store a pointer to the agent's circular buffer. The end result is that the replica must read the pointer to the circular buffer indirectly, through a gs-relative memory access. In assembler, we manually crafted a version of our WoC agent that accesses this pointer in such a way. By storing the pointer to the buffer and any pointers derived from it in a fixed caller-saved general-purpose register, we guarantee that the pointer never leaks to memory and that no function outside the replication agent can observe the pointer value. We further guarantee that (i) the pointer to the buffer is never moved to a different register, (ii) the register is never pushed onto the stack, (iii) the register is cleared before the function returns and (iv) the replication agent does not call any functions while the pointer value is visible.

Since neither gcc, nor LLVM offers syntactic sugar to allow for such properties, we have chosen to implement both of the replication agent's functions that access the shared buffer in assembly code. The current implementation, which we evaluate in Section 4.2, totals approximately 150 LoC.

4.1.4 Embedding the replication agent

The key challenge in embedding the replication agent into a program is to identify all sync ops. Because we want to wrap these sync ops in the source code itself, we also identify the sync ops at the source code level.

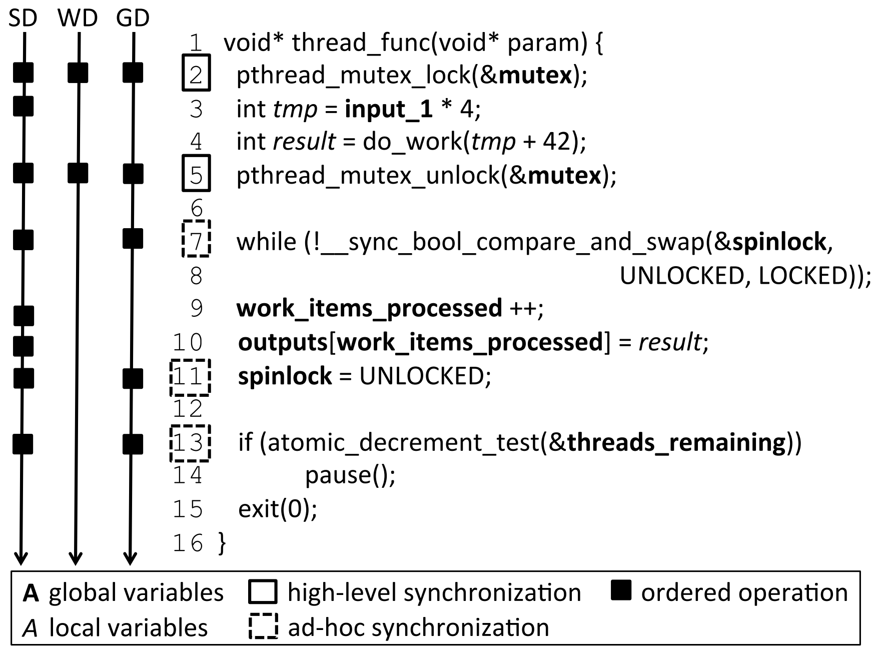
Existing R+R systems, as well as DMT systems that impose weak determinism, only order invocations of pthread-based synchronization functions. That is insufficient for a secure MVEE because glibc and several other low-level libraries implement their own sync ops. A failure to order the sync ops in glibc tends not to affect the user-observable I/O determinism of the program, but it does impact the general system call behavior and hence violates the system call consistency needed in a secure MVEE.

An alternative strategy would be to order all sync ops by wrapping all loads and stores in the program. This would yield system call consistency even in the presence of data races. Ordering individual loads and stores leads to prohibitively high overhead however, as was demonstrated in the context of strong determinism [40]. Moreover, given the range of diversity we need to support in the replicae to mitigate a sufficiently wide range of attacks, there is no guaranteed one-to-one mapping between the loads and stores in the different replicae.

The strategy we propose is most similar to weak determinism systems, but we capture sync ops at a lower level as shown in Figure 4.4. In existing strong determinism systems, all 11 memory operations need to be ordered. (Lines 3, 9, and 10 each involve two memory operations.) With existing systems of weak determinism, only the two (standard synchronization) operations on the mutex on lines 2 and 5 are ordered. In our solution, we wrap both the standard operations on the mutexes as well as the three ad hoc synchronization events on lines 7, 11 and 13. The latter one translates into a LOCK SUB instruction on the x86 architecture, and atomically sets the zero flag (ZF).

To identify the source lines to be wrapped, we first identify the sync ops in the binary code, and then translate those to source line numbers by means of debug information.

The relevant sync ops in the binary code come in three categories. First, any instructions that the programmer explicitly marks as atomic are sync ops. On the x86 architecture, this includes all instructions with an explicit LOCK prefix, as well as XCHG instructions with an implicit



```
void* thread_func(void* param) {
  pthread_mutex_lock(&mutex);
  int tmp = input_1 * 4;
  int result = do_work(tmp + 42);
  pthread_mutex_unlock(&mutex);

  while (!__sync_bool_compare_and_swap(&spinlock,
                                        UNLOCKED, LOCKED));
  work_items_processed ++;
  outputs[work_items_processed] = result;
  spinlock = UNLOCKED;

  if (atomic_decrement_test(&threads_remaining))
        pause();
  exit(0);
}
```

Figure 4.4: Comparison between three classes of determinism.

LOCK prefix [58]. LOCK CMPXCHG is an example. Second, any store operation (e.g., for a C assignment to a dereferenced pointer or volatile variable) that directly succeeds an explicit memory barrier is a sync op. Such stores are typically used in synchronization schemes like read-copy-update (RCU). Third, any instruction that references a memory-address (such as some memory-allocated variable) that is referenced by another sync op also becomes a sync op. We refer to these operations as *unprotected loads and stores*, a terminology sometimes used to denote benign data races.

The guarantees we provide by running a replication agent that enforces an equivalent order of sync ops in all replicae are at least as strong as the guarantees that weak determinism provides. To see why, recall that weak determinism enforces a deterministic order of entries into critical sections. Thus, in a correct program, weak determinism will grant mutually exclusive access to related blocks of shared memory in a deterministic order. Though this was historically not the case [42], all modern user-mode programs that run on SMP-systems now implement mutual exclusion using an atomic test-and-set operation [68, 88]. On x86 systems, several instructions provide test-and-set semantics, but they are atomic only if a LOCK prefix is used. Consequently, the first category contains all instructions that may implement the entrance into a critical section as a sync op. Operations that implement the exit from a critical section can either be implemented using atomic test-and-set or exchange operations (second category) or with atomic store operations (first or third category).

On the GNU/Linux platform, all high-level synchronization primitives in the pthreads, OpenMP and C++ standard libraries are based on the mutual exclusion principle. In the pthreads library, e.g., functions such as pthread_mutex_lock and pthread_cond_wait implement mutual exclusion using LOCK CMPXCHG instructions. In the implementation of other high-level synchronization primitives, a variety of atomic test-and-set operations are used. All of them are prepended with LOCK prefixes, however, and all of them are therefore classified as sync ops.

The identification and wrapping of sync ops is currently a partially manual process. First, we disassemble the binary/library to identify explicit memory barrier instructions or instructions with explicit or implicit LOCK prefixes. If no such instructions are present, no further steps are needed. We use debugging symbols to map all of the identified instructions to their originating source line. For memory barriers, we wrap the store that directly succeeds the barrier in calls to our replication API. We identify loads of the same variable and wrap them too. To some extent, we thus fix benign and deliberate data races, such as when developers use such a barrier to set a flag without synchronization.

For source lines that compile into instructions with LOCK prefixes, check whether or not compiler intrinsics are used to express the atomic operations. If not, we insert calls to our replication API before and after the operation. Otherwise, we include an automatically generated header in the source file. This header overrides all known intrinsics that implement atomic operations and inserts the appropriate replication API calls automatically. It is generated by a simple script that downloads the list of all atomic compiler intrinsics from the GNU GCC website, parses the list and generates the necessary definitions. In addition, we identify other loads and stores of the variables involved in the atomic operations and insert API calls manually.

While this process might seem cumbersome, it is important to note that unless a program or higher-level library implements its own ad hoc synchronization or lock-free algorithms, no memory barriers or instructions with LOCK prefixes will be found. So most programs are supported transparently. For our tests, only four libraries needed modifications. We report extensively on the size of these modifications in Section 4.2.1.

In many cases, and in particular in portable code, the manual effort to invest in these modifications is very limited. In portable code, compiler intrinsics are used to implement atomic operations, as well as all other accesses to the variables involved in those atomic operations. This is necessary to ensure portability to architectures that do not guarantee the atomicity of aligned loads and stores. To wrap all the necessary compiler intrinsics, it suffices to include our automatically generated header in all source files.

Furthermore, even for programs and libraries that do need more modifications, the patching process can be streamlined to a great extent. C++11 compliant compilers provide a template for atomic operations [13]. With this template, a programmer can mark variables that need to be updated atomically by modifying their type, i.e., by wrapping the type in the std::atomic template. During the compilation, the compiler translates all accesses to such variables such that the appropriate atomic intrinsic is used for every access. Our automatically generated header can then insert the appropriate calls to our replication API automatically by overriding these intrinsics. In summary, if we use a C++11 compliant compiler, the patching effort can be limited to modifying the type of all variables that need to be accessed and updated atomically. In our future work, we plan to extend our current embryonic implementation in LLVM to automate this process completely.

The replication-enabled libraries can replace their original counterparts, in which case they will function correctly and with minimal overhead outside the MVEE context. Alternatively, they can be installed side-by-side with the original ones, in which case the MVEE will intervene transparently in the library loading to load the replication- enabled ones. Our solution hence places a minimal burden on system administrators and users.

4.1.5 Interaction with the Kernel

Synchronization algorithms often rely on the kernel's futex API to interact with other threads and processes. The multi-purpose synchronization API is exposed through a system call, and is used throughout GNU's pthreads library for two reasons. First, some functions use the

FUTEX_WAIT operation to block the calling thread until the value stored at a specified address changes. In the pthread_mutex_lock function, this operation is used to block the calling thread if the mutex is currently contended. Other functions such as pthread_cond_wait use the FUTEX_WAIT operation to block until an event occurs. Other functions use the FUTEX_WAKE or FUTEX_CMP_REQUEUE operation to signal threads that are waiting for the value at the specified address to change. In the pthread_mutex_unlock function, wake operations are used to wake up threads that are blocked in a related pthread_mutex_lock call. Functions such as pthread_cond_signal or pthread_cond_broadcast use wake operations to signal and wake up threads that are blocked inside a related pthread_cond_wait call.

A potential issue arises when a thread performs a wake operation for which only one other thread should be woken up. If more than one thread is waiting in each replica and the replication agent does not intervene, the kernel might wake up non-corresponding threads in the replicae. In a slave replica, the woken thread will then stall indefinitely (i.e., deadlock) at the first atomic op it encounters.

While this issue can be handled in the replication agents embedded in all replicae, we chose to tackle the issue from within GHUMVEE's monitor instead. The monitor allows all replicae to invoke futex calls but it allows only the master replicae to actually complete the call. The monitor manipulates the system call number and arguments for the slave replicae's futex calls to have them perform a harmless non-blocking system call instead (such as sys_getpid). The non-blocking call returns immediately from the kernel, at which point the monitor stalls the slave until the master's futex call has also returned. At that point, the monitor replicates the result of the system call to all slaves and resumes them. This guarantees that the corresponding threads get woken up in all replicae. By implementing the logic in the monitor instead of in the agent, we keep the agent small and fast for its other replication tasks. The additional overhead of going through the monitor is relatively small, given that it already intercepts all system calls invocations and returns anyway.

4.2 Evaluation

4.2.1 Embedding the replication agent

To evaluate the run-time overhead of GHUMVEE, we used the PARSEC 2.1 benchmark suite. We did not include the facesim and canneal benchmarks because they contain many data races. For canneal, this is hardly surprising as it is based on data race recovery. We applied minor patches[1] to four benchmarks to eliminate data races or to embed our agent. ferret raced on the cnt_enqueue and input_end variables. Additionally, the imagick library on which ferret depends contained unprotected accesses to the free_segments variable. freqmine raced on the thread_begin_status variable. raytrace used ad hoc synchronization in its AtomicCounter class. vips had an unprotected read and write in the gclosure.c file.

We further applied fixes for bugs which had been reported in the literature or on the PARSEC web site. We configured fluidanimate and streamcluster benchmarks to use the original pthread-based barriers rather than the semantically equivalent but less efficient parsec-based barriers.

On our testing system, the benchmark suite relies on four libraries in which we embedded our replication agent: glibc 2.19, libpthread 2.19, libstdc++ 4.8.2 and libgomp 4.8.2, the default library versions of Ubuntu 14.04. Since libpthread and glibc are built from the same source tree, we treat them as one entity when reporting the required patching effort.

For libgomp and libstc++, we leverage the use of compiler intrisics as discussed in Section 4.1.4. Both libraries support specialized targets and more generic targets: libstdc++ supports the so-called i486 and generic CPU targets, while libgomp supports the so-called Linux and POSIX targets. We adapted the makefiles, directory structures, and linker scripts to ensure that the code targeting the generic CPU and POSIX are used instead of the code in support of the more specific i486 and Linux targets, thus ensuring that code relying on compiler intrisics is used instead of code involving inline assembly. Furthermore, we made sure that the automatically generated header was included in all relevant source files.

For each of the libraries, all of this preparation required editing/executing less than 14 lines of script and source code. The automatically generated header consists of 131 LoC. In addition, in libgomp, 2 lines of

[1]At `http://ghumvee.elis.ugent.be` our patches, raw data and scripts are available. GHUMVEE will be open sourced in Q4 2015.

code needed to be edited to replace two unprotected load/store operations by atomic ones. So all in all, a very limited effort was required to prepare these libraries: 2*14+2=30 lines of code needed to be edited manually to prepare the two libraries that total about 110k LoC. Moreover, this manual editing was limited to 4 source files out of a total of 673 files.

A considerably larger patching effort was needed to embed our agent in glibc/libpthread, because they use ad hoc synchronization throughout and have many explicit memory barriers and unprotected loads and stores.

Whereas more modern glibc/libpthread ports like the ARM port use compiler intrinsics to implement their atomic operations, the AMD64 and i386 ports do not, presumably because intrinsic support in compilers was not up to par yet when those ports were developed. Instead, the AMD64 and i386 ports rely on inline assembler. With today's compiler support for intrinsics, the inline assembler can be replaced by intrinsics without performance penalty. In addition, our effort for embedding the replication agent in glibc would have been much reduced in case the inline assembler had already been replaced by the intrinsics. As this is not yet the case, we needed to do the replacement ourselves. For this purpose, we replaced the x86 version of the lowlevellock.h header by the ARM version of that same file. We also deleted the assembly-based x86-specific versions of many pthread functions from the source tree, such that the generic versions of those same functions are used instead.

This did not suffice, however. Contrary to libgomp and libstc++, glibc's generic code does not use compiler intrinsics directly. Instead, glibc implements its own series of sync ops, of which some map directly to compiler intrinsics and others do not. So we opted to wrap glibc's sync ops manually, rather than with the automatically generated header. Our manually constructed wrappers span 211 lines of code. We added an additional 175 lines of code to allow ld-linux, which is also built from glibc's source tree, to still use the original unwrapped macros as needed by GHUMVEE. We therefore needed 386 lines of code in total to wrap all synchronization operations in glibc-libpthread. In addition, we added approx. 261 lines of code to eliminate data races.

Finally, we added 14 lines in one of glibc's linker scripts to expose our replication agent to other libraries, and added our replication agent itself. The WoC agent, for example, counts no more than 194 lines of C code, while the secure WoC agent counts 167 lines of assembly code and 100 lines of C code.

Excluding the copying and deleting of existing code, as well as our own replication agent, the source code patching effort to prepare glibc/libpthread was limited to 211+175+261+14 = 661 LoC in 60 source code and build files. Compared to the library's total size of several 100K LoC spread over several thousand source code files, this effort is still fairly limited. And all of it can of course be reused for replicating all applications.

Moreover, since version 2.20, a gradual effort is ongoing in the glibc developer community to replace inline assembler sync ops by their more portable, more generic, more maintainable counterparts in the form of compiler intrinsics. Together with the automated support we are developing as mentioned near the end of Section 4.1.4, this will reduce the required patching effort significantly in the near future.

4.2.2 Run-time overhead and scalability

We evaluated our technique on a system with two Intel Xeon E5-2650L processors with 8 physical cores and 20MB cache each. The system has 128GB of main memory and runs the AMD64 version of the Ubuntu 14.04 OS. For the sake of reproducibility, we disabled hyper-threading and all power saving and dynamic frequency and voltage scaling features. The system runs a Linux 3.13.9 kernel that was compiled with a 1000Hz tick rate to minimize the monitor's latency in reacting to system calls. We applied a small optional kernel patch (less than 10 LOC) that adds a variant of the sys_sched_yield system call that bypasses GHUMVEE. Other than that, no kernel patches were used. With this small kernel patch, our agents can efficiently yield the CPU whenever they are waiting for preceding sync ops to finish replaying in the slave replicae. This patch improves the performance of our TO and WoC agents in the dedup benchmark but has no significant effects elsewhere.

All benchmarks were compiled at optimization level -O2 using GCC 4.8.2. The native performance of the benchmarks was measured using the original, unpatched libraries that shipped with the OS. GHUMVEE performance was measured using their GHUMVEE-enabled versions.

We measured the execution time overhead of our agents by running each PARSEC benchmark with 1 to 8 worker threads natively as well as in GHUMVEE with 2, 3, and 4 replicae. Using the native PARSEC input sets, i.e., the largest standardized set, we ran each measurement five

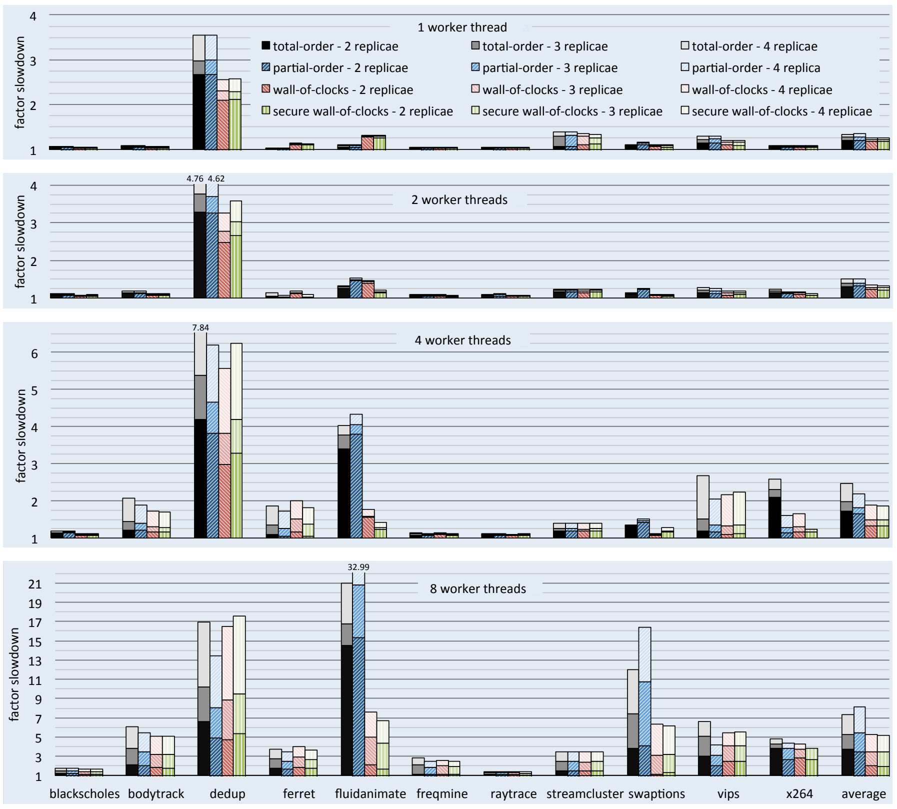


Figure 4.5: GHUMVEE execution time overhead, relative to native execution of PARSEC 2.1 applications for different numbers of worker threads. Each stack for each benchmark shows the overhead for 2, 3, and 4 replicae. The four stacks per benchmark correspond to the three (non-secured) agents + the secure version of the WoC agent.

times, of which we omitted the first to account for I/O-cache warmup. For 1, 2, 4, and 8 worker threads, Figure 4.5 presents the benchmarks' execution time as replicated by GHUMVEE, relative to the native versions. For each agent and for each benchmark, Figure 4.6 shows how the native and the replicated execution (for two replicae) scales with the number of worker threads.

These figures display several trends. Most importantly, with both WoC agents, many benchmarks (blackscholes, freqmine, raytrace, swaptions, and even streamcluster) can be replicated with little overhead up to 8 worker threads, and in some cases even with 3 or 4 replicae. Other benchmarks (bodytrack, ferret, vips, x264) can be replicated with little over-

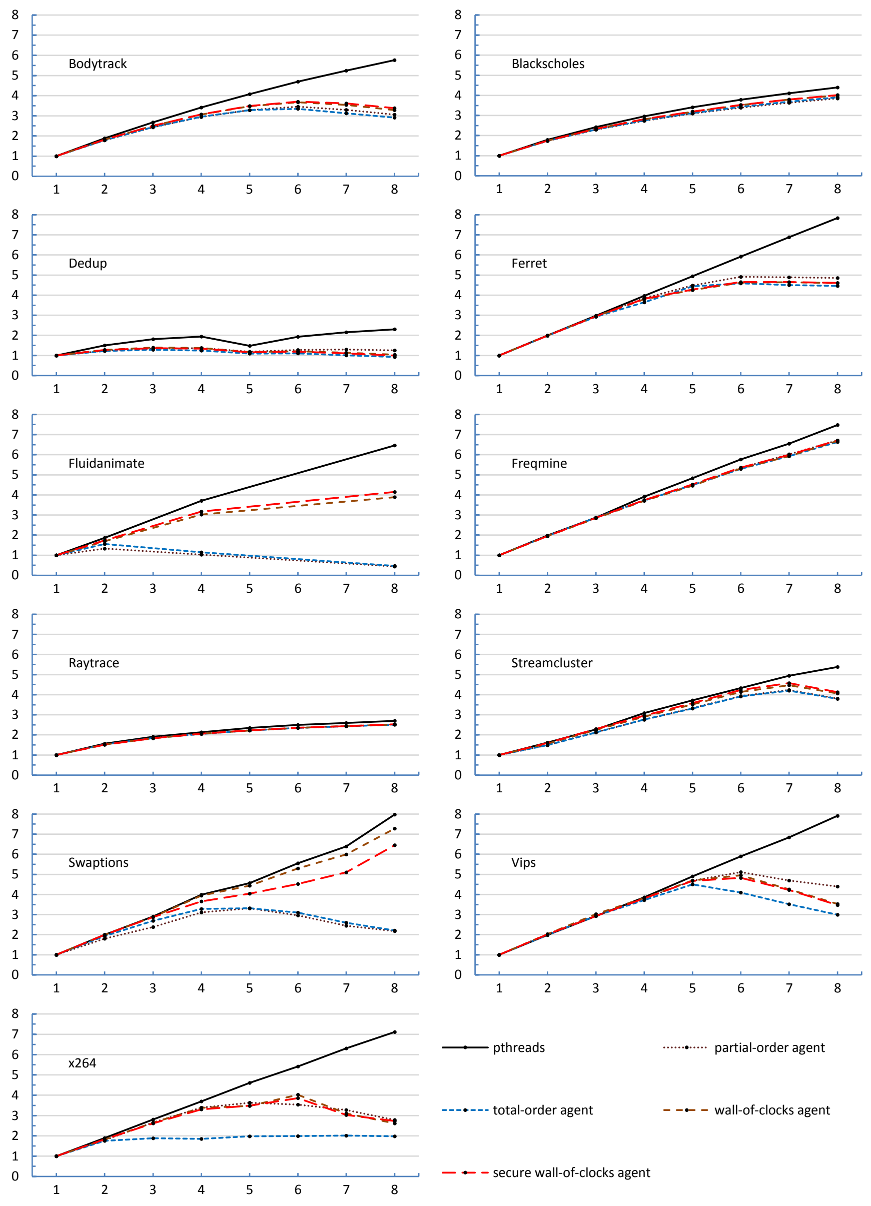


Figure 4.6: Scaling of the native (i.e., pthreads) and replicated benchmarks (for two replicae, one master and one slave) with the four different replication agents. On the X-axis, the number of worker threads is given. On the Y-axis, the performance relative to one worker thread is presented. Fluidanimate only runs when the number of worker threads is a power of two.

head up to 4 worker threads. The average overhead of the replication remains below 2x for 2 replicae with the WoC agents, even with 8 worker threads. With more replicae, the overhead clearly increases. This is of course the result of resource contention of the many threads over the limited number of cores. For most of the mentioned benchmarks, it then does not matter too much which agent is used.

For other benchmarks, however, there is a big difference in overhead between the different agents, and there are several benchmarks for which significantly larger overheads and bad scaling are observed.

First, regardless of which agent we use, dedup consistently suffers high performance penalties. The main contributor to this overhead is the high system call density in dedup. When running with 8 worker threads, dedup executes over 123k system calls/second. This density is far greater than in any other program we have tested so far. In the PARSEC suite itself, the highest density we have measured besides dedup was for the vips benchmark (20.9k system calls/second for 8 worker threads). In older benchmark suites such as SPEC CPU2006, the highest density we have measured was around 1k system calls/second for 403.gcc. The high overhead in benchmarks with such high system call densities is unfortunately a fundamental problem of the ptrace API on which GHUMVEE and most other security-oriented MVEEs rely to monitor the behavior of the replicae.

Second, the swaptions and fluidanimate benchmarks, which use fine-grained synchronization, expose a major bottleneck in our PO and TO agents. Both of these benchmarks use a fork/join threading model and frequent, fine-grained synchronization. In both of these benchmarks, all worker threads perform the same tasks and progress at roughly the same pace. While swaptions does not use any explicit synchronization in the application code itself, it does rely heavily on dynamic memory allocation. The dynamic memory allocator in GNU's libc uses ad hoc synchronization and lock-free algorithms to ensure thread safety. Through libc, swaptions executes more than 398M sync ops when running with 5 worker threads. With 8 worker threads, swaptions performs over 403M sync ops. This corresponds to 4.2M sync ops per second in the native benchmark with 5 worker threads, and up to 7.5M sync ops per second in the native benchmark with 8 worker threads.

In fluidanimate, the situation is even worse. Contrary to swaptions, fluidanimate does invoke our replication agent directly. With 4 worker threads, fluidanimate performs over 1.18B sync ops, which corresponds to over 9.8M sync ops per second in the native benchmark. These sync ops originate mainly from the pthread_mutex_lock and pthread_mutex_unlock functions, which are used to acquire or release one of the 2.31M individual mutexes used in the program. With 8 worker threads, fluidanimate performs over 2.35B sync ops, which corresponds to over 32.9M sync ops per second in the native benchmark. Because the lock and unlock operations are spread over so many different mutexes, there is very little contention in the native benchmark.

In GHUMVEE, however, the TO and PO agents create a lot of contention. Both agents capture the total order of the sync ops in a single circular buffer. To capture this order, the agents acquire a lock before executing the original atomic op and do not release this lock until the operation has been logged in the buffer. These agents therefore effectively serialize the execution of sync ops in the master replica.

A second problem with these two agents is that all replicae must keep track of their current position in the buffer. This position must be read before the processing of each atomic op, and updated after each atomic up. Every time an update happens on a different core that does not share a cache with the core on which the previous update was executed, the cache line that contains the current position in the circular buffer will be invalidated, and hence cause stalls in the cores' pipelines.

The combination of these two bottlenecks results in poor scaling for swaptions and fluidanimate. In other benchmarks, the effects of the serialization and the additional cache coherence traffic incurred by our TO and PO agents are less visible. The main reason is that the other benchmarks perform much less sync ops than swaptions and fluidanimate. In the other benchmarks, the highest sync op rates occurred for dedup and vips, with 936K and 644K sync ops per second in the native benchmark resp.

Most importantly, our WoC agents almost completely eliminate the bottlenecks observed in the swaptions and fluidanimate benchmarks. Only in the vips benchmark, these agents cannot avoid a significant serialization overhead when the number of worker threads increases, because it assigns many unrelated mutexes to the same logical clocks. Thus, for this specific benchmark, the PO agent outperforms the WoC agents.

A final trend is that benchmarks that use condition variables do not scale well beyond 6 worker threads. The bodytrack, dedup, ferret, vips and x264 benchmarks all rely on condition variables to signal and to wake up threads. With enough available CPU time, all of the benchmark's threads can run simultaneously and a thread can be signaled without resorting to sys_futex calls. The five mentioned benchmarks all have heterogeneous threading models and have more than n threads running simultaneously (with n the number of worker threads). For that reason, our machine's 16 cores simply cannot run all threads in 2 or more replicae simultaneously.

All in all, the WoC agents perform reasonably well. With 4 worker threads and two replicae, the average slowdown of our MVEE is only 1.33x with the regular WoC agent and 1.32x with the secured WoC agent. With the TO and PO agents, the average slowdown is 1.73x and 1.64x resp. The high system call overhead in the dedup benchmark is the main contributor to the slowdown. In this configuration, the slowdown in dedup ranges from 2.98x with the WoC agent to 4.19x with the TO agent.

With 8 worker threads and two replicae, the average slowdown is much higher. The slowdown for our TO, PO, WoC and secured WoC agents is 3.69x, 3.42x, 1.99x, and 1.98x resp. For our TO and PO agents, the main cause is the introduced serialization and constant cache invalidations that come with the single circular buffer approach. Our WoC agents eliminate this bottleneck for the most part, but in this configuration, the lack of resources on our test machine becomes a problem. The variations in results for the WoC agents are caused by minor differences in their implementation. The regular WoC agent accesses the synchronization replication buffer directly, whereas the secured WoC agent accesses the buffer indirectly, as we explained in Section 4.1.3. This indirection slightly increases the cache pressure. As opposed to the regular WoC agent's C implementation however, the secured WoC agent's hand-written assembler implementation does not spill any registers to the stack. This optimization slightly reduces the cache pressure. The combination of these two minor implementation differences slightly favors the secured WoC agent in terms of performance.

4.2.3 Security Evaluation

All of our replication agents rely on a buffer that is shared between all replicae. This buffer is mapped as a read-write memory segment in the master replica and as a read-only segment in the slave replicae. Intuitively, it might seem like a security risk to create such a communication channel between the replicae because it can be used to forward information from the master to the slave replicae without triggering a monitored RVP. In practice, however, the security risk is minimal.

In principle, it is possible to launch attacks that cause the master replica to write arbitrary data into the buffer, but the master replica cannot instruct the slave replicae to use the arbitrary data in any meaningful way other than to replay synchronization operations, because the data written into the synchronization buffer is only read by the replication agent. We have manually audited our replication agents and verified that they never pass any information they retrieve from the synchronization buffer on to any other part of the program. GHUMVEE further ensures that explicit input, i.e., input retrieved from system calls, is never written into the synchronization buffer.

We do, however, anticipate future MVEE designs in which the MVEE does not arbitrate all system calls that may retrieve input. For example, the recently proposed reliability-oriented VARAN handles system call monitoring and input replication entirely in user space and inside the context and address space of the replicae [53]. In such a system, the synchronization buffer could in theory be used as an uncontrolled communication channel, which might aid a compromised master replica in mounting an attack on the slave replicae. Specifically, the compromised master replica could manipulate the return values of its system calls, thereby instructing the slave replicae to read further input from the synchronization buffer.

To protect the synchronization replication buffer in this scenario, GHUMVEE forces the buffer to be mapped at different, randomized addresses in each replica. A compromised master replica therefore would not know the exact location of the buffer in the slave replicae and it would have to derive the location through **information leakage** or by **guessing**. Alternatively, the master replica could try to construct a **code reuse attack** that invokes the replication agent's code to read from the synchronization buffer.

GHUMVEE prevents the latter attack with its DCL. The master replica can therefore not mount a code reuse attack: He cannot assume that slave replicae have the same memory layout as the master replica, and if he feeds an address to the replicae that points to an executable gadget in the master's address space, the slave will raise an exception when it tries to execute code at the same address.

Guessing the location of the buffer is hard. GHUMVEE currently use synchronization buffers of $256MiB$, which corresponds to 65536 memory pages. The AMD64 ABI allows user-space applications to use

48 bits for memory addressing but excludes the first memory page, i.e., the page that starts at address $0x0$. Therefore, a user-space application may map up to $2^{36} - 2$ memory pages. The chance to blindly guess the location of a $256MiB$ buffer in one slave replica is therefore $65536/(2^{36} - 2)$ or $9,53 \cdot 10^{-7}$.

Forcing the slave replicae to leak the location of their synchronization buffer is not trivial either. While the TO, PO and regular WoC agents do internally store a pointer to the synchronization buffer, GHUMVEE prevents leakage of the pointer from the slaves to the master through the buffer by mapping the buffer as read-only in the slaves. As GHUMVEE intercepts all system calls, it is trivial to prevent a replica from reverting that memory protection.

Leaking the pointer through other channels is still possible, however. We have therefore constructed our secured WoC agent, which significantly reduces the odds of a successful leakage attack. This secured agent does not store a pointer to the synchronization buffer, but instead accesses the synchronization buffer only through an indirection via the `gs` segment, as explained in Section 4.1.3.

A remaining option to consider is the use of a covert channel between the replicae, and to use the MVEE as the medium through which the covert channel communicates. For example, the replicae can deliberately delay each other by exploiting the MVEE's lock-step execution mechanism. This mechanism dictates that certain operations may only be completed when all replicae attempt to invoke them. The length of the delay can represent information such as individual bits of a pointer value. While it is easy to write programs that intentionally set up and exploit such channels, it is not possible to deploy this technique if the MVEE's protection policy is properly implemented. In GHUMVEE, DCL prohibits the launch of a code reuse attack (to setup and exploit the covert channels) in the first place.

In conclusion, we believe that our synchronization replication buffers and agents are sufficiently hardened against attacks, even in scenarios where the master replica can forward explicit input to the slave replicae via those buffers.

4.3 Related Work

4.3.1 MVEEs

Throughout the last decade, several MVEEs have been presented. Cox et al. first proposed N-Variant Systems, a kernel-space MVEE [35]. Shortly afterwards, Cavallaro presented a proof-of-concept user-space MVEE [27]. Salamat et al. then proposed Orchestra, a more advanced user-space MVEE [114]. More recently, Hosek and Cadar presented Mx [52] and VARAN [53], while Maurer and Brumley introduced Tachyon [85]. The latter three systems are not security-oriented MVEEs like the former ones, as they aim at safe testing of experimental software updates, rather than at protecting programs against exploits. The only multi-threaded applications on which VARAN was tested were server applications in which none of the system call behavior depends on the thread synchronization order: Those server benchmark threads perform almost completely independent computations. By contrast, the system call behavior in the PARSEC benchmarks, even our data race free versions, depends very much on the synchronization order. Without replicating and ordering synchronization events, none of the PARSEC benchmarks can be handled correctly. VARAN's approach of ordering system calls and signals but not synchronization events, is simply not a generic, reliable solution. While some of the existing MVEEs have been evaluated on multi-process applications (such as older version of Apache), GHUMVEE is the first to provide active support for non-deterministic, multi-threaded applications.

4.3.2 Deterministic Multithreading

Deterministic MultiThreading (DMT) systems impose a deterministic schedule on the execution order of instructions that participate in inter-thread communication, or a deterministic schedule on the order in which the effects of those instructions become visible to other threads. Some DMT systems guarantee determinism only in the absence of data races (*weak determinism*), while others work even for programs with data races (*strong determinism*).

Some DMT implementations, especially the older ones, rely on custom hardware [12, 40, 41, 54] or a custom operating system [6, 16]. Of interest to us, however, are the user-space software-based approaches [10,15,17,37,40,81,82,89,97,108,145].

Software-based DMT systems come in many flavors but essentially, they all establish a deterministic schedule by passing a token. We refer to the literature for an excellent overview of the possible ways to implement the deterministic schedule, as well as their implications [119]. In the remainder of this discussion, we focus on the fundamental reason why DMT systems are incompatible with MVEEs that run diversified replicae: the timing of and prerequisites for the deterministic token passing.

Most DMT systems require that all threads synchronize at a global barrier before they can pass their token. Some of the systems that employ such a global barrier, insert calls to the barrier function only when a thread executes a synchronization operation [10, 17, 81, 108]. This approach is incompatible with parallel programs in which threads deliberately wait in an infinite loop for an asynchronous event such as the delivery of a signal to trigger. Such threads never reach the global barrier. Other DMT systems tackle this issue by inserting barriers at deterministic points in the thread's execution. These deterministic points are based on the number of executed store instructions [97], the number of issued instructions [145] or the number of executed instructions [15,40]. All of these numbers are extremely sensitive to small program variations, which makes such systems an ill fit for use in diversified replicae.

Conversion [89] does not use a global barrier but, like other DMT systems, it relies on a deterministic token that can only be passed when threads invoke synchronization operations, which again is incompatible with parallel programs in which some threads never invoke synchronization operations. RFDet [82] uses an optimized version of the Kendo algorithm [97] to establish a deterministic synchronization order. Like Kendo however, the order is still based on the amount of executed instructions in each thread, which makes RFDet equally sensitive to program variations.

4.3.3 Record/Replay

Record/Replay (R+R) systems capture the order of synchronization operations in one execution of a program and then enforce the same order in a different execution. This can happen offline, by capturing the order in a file to be replayed during a later execution of the same program, or online, by broadcasting the order directly to another running instance of the program. In the absence of data races, R+R systems show many similarities with DMT techniques that impose weak determinism.

RecPlay is a prime example of an offline R+R system [110]. During recording, RecPlay logs Lamport timestamps for all pthread-based synchronization operations [67]. During subsequent replay sessions, synchronization operations are forced to wait until all operations with a earlier timestamp have completed. Because it only enforces the order of synchronization operations, RecPlay's replication mechanism incurs less overhead than preexisting techniques that replicate the thread scheduling order or the order in which interrupts are processed [5]. Moreover, RecPlay assigns the same timestamp to non-conflicting synchronization operations, such that they can also be replayed in parallel.

Loose Synchronization Algorithm (LSA) was one of the first techniques that adopted R+R for use in fault-tolerant systems [11]. LSA designates one of the nodes as the master node. The master node records the order of all pthread-based mutex acquisitions and periodically replicates this order to the slave nodes. These slave nodes then enforce the same acquisition order on a per-mutex basis.

More recently, Lee et al. proposed Respec online replay on multiprocessor systems [69]. Oriented towards fault-tolerant execution of identical replicae, Respec purposely records an unprecise order of synchronization operations in the master process and speculatively replays that order in the slave processes. At the end of a replay interval, Respec checks whether the slaves are still synchronized with the master process by comparing their state, incl. their register contents. If not, it rolls them back. While recording, Respec maps synchronization variables onto a statically allocated clock, similarly to our WoC agents. It is doubtful, however, whether Respec's approach could work in a security-oriented MVEE like ours, in which diversity in the replicae makes it hard (if not impossible) to detect whether the replicae have diverged at the end of a replay interval.

Other online R+R techniques rely on custom hardware support [12], and hence are not useful for a secure MVEE for off-the-shelf systems.

4.4 Conclusions

This chapter presented how GHUMVEE was extended to become the first security-oriented MVEE that can replicate parallel programs correctly. We proposed three replication strategies and implemented four replication agents to implement them, one of which does so over a secured communication channel. Our replication agents are conceptually similar to existing tools, but unlike existing tools, they fit within the constraints that a security-oriented MVEE imposes for lock-step monitoring of diversified replicae.

Additionally, we proposed a new strategy to embed a replication agent into parallel programs, incl. programs that use ad hoc synchronization primitives, and we evaluated the effort to do so. In the future, we plan to automate this strategy to a large degree.

We extensively evaluated the effect of our MVEE and our replication agents on the PARSEC 2.1 benchmarks on the GNU/Linux platform. With our secure wall-of-clocks agent, the best of the four agents, we achieve an average slowdown of just 1.32x when running the benchmarks with 4 worker threads and 2 replicae.

Chapter 5

Disjoint Code Layouts

In 2007 Shacham presented the first Return Oriented Programming (ROP) attacks for the x86 architecture [121]. He demonstrated that ROP attacks, unlike return-to-libc attacks, can be crafted to perform arbitrary computations provided that the attacked application is sufficiently large. ROP attacks were later generalized to more architectures such as SPARC [24], ARM [65], and many others. Despite the progress and activity on the attacker front, defense against ROP attacks is still very much an open problem. As will be discussed in the related work section, all of the existing solutions come with important drawbacks and limitations.

As an alternative protection against user-space ROP attacks, we present Disjoint Code Layouts (DCL). This technique relies on the execution and replication of multiple run-time variants of the same application under the control of a monitor, with the guarantee that no code segments in the variants' address spaces overlap. Lacking overlapping code segments, no code gadgets co-exist in the different variants to be executed during ROP attacks. Hence no ROP attacks can alter the behavior of all variants in the same way. By monitoring the I/O of the variants, and halting their execution when any divergent I/O operation is requested, the monitor effectively blocks any ROP attack before it can cause harm. Our design and implementation of DCL offers many advantages over existing solutions:

- DCL offers immunity against ROP attacks, rather than just raising the bar for attackers.
- The execution slowdown incurred by our monitor and protection is minimal, up to orders of magnitude smaller than in some exist-

ing approaches.

- A single version of the application binary suffices to protect against ROP attacks. Optionally, our monitor supports the execution and replication of multiple diversified binaries of an application to also protect against other types of memory exploits.

- With user-space tools only, we achieve complete immunity against user-space application ROP attacks. No adaptation of the underlying Linux OS is needed.

- Similarly, our solution only requires run-time intervention. It is therefore compatible with existing compilers and existing solutions such as stack protectors.

- Requiring no or only trivial adaptations of the software by the developer to make his application support our monitor's replication, the presented techniques are applicable to a wide range of applications, including multi-process multi-threaded applications that rely on custom synchronization libraries and that feature address-dependent behavior.

- Unlike some existing techniques, DCL causes only marginal memory footprint overhead within the protected application's address space. Thus, DCL can protect programs that flirt with address space boundaries on, e.g., 32-bit systems. System-wide, DCL does cause considerable memory overhead due to its duplication of process-local data regions such as the heap and writable pages. Still, DCL outperforms memory checking tools in this regard.

Combined, these features of our multi-variant execution engine design make this form of strong protection much more convenient to deploy than existing state of the art.

5.1 Completely Disjoint Code Layouts

Our technique of Disjoint Code Layouts (DCL) is implemented mostly inside GHUMVEE's monitor. Its support for DCL is based on the following Linux features:

- In general, any memory page that might at some point contain executable code is mapped through a sys_mmap2 call. When the program interpreter (e.g., ld-linux) or the standard C library (e.g., glibc) load an executable or shared library, the initial sys_mmap2 will request that the entire image be mapped with PROT_EXEC rights. Subsequent sys_mmap2 and sys_mprotect calls then adjust the alignment and protection flags for non-executable parts of the image. Section 5.1.1 discusses the few exceptions.
- Even with ASLR enabled, Linux allows for mapping pages at a fixed address by specifying the desired address in the addr argument of the sys_mmap2 call.
- When a replica enters a system call, this constitutes a RVP for GHUMVEE, at which GHUMVEE can modify the system call arguments before the system call is passed on to the OS. Consequently, GHUMVEE can modify the addr arguments of all sys_mmap2 calls to control the replica's address space.

As shared libraries are loaded into memory from user space, i.e., by the program interpreter component to which the kernel transfers control when returning from the sys_execve system call used to launch a new process, GHUMVEE can fully control the location of all loaded shared libraries: It suffices to replace the arguments of any sys_mmap2 call invoked with PROT_EXEC protection flags and originating from within the interpreter. Some simple bookkeeping in the monitor then suffices to enforce that the code mapped in the different replicae does not overlap, i.e., that whenever one variant maps code onto some address in its address space, the other ones do not map code there.

Some code regions require special handling, however. Under normal circumstances the kernel maps those regions. But because GHUMVEE cannot intervene in decision processes in kernel space, it needs to prevent the kernel from mapping them and instead have them mapped from user space instead, i.e., by the program interpreter. GHUMVEE can then again intercept the mapping system calls, and enforce non-overlapping mappings of code regions.

5.1.1 Initial Process Image Mapping

The standard way to launch new applications is to fork off a running process and to invoke a sys_execve system call. For example, to read a directory's contents with the ls tool, the shell forks and invokes sys_execve("/bin/ls", {"ls", ...}, ...); The kernel then clears the virtual address space of the forked process and maps the following components into its now empty address space as depicted in Figure 5.1.

An initial stack is set up first. With ASLR enabled, the stack base is subject to randomization. As we mentioned before, only bits 12 through 27 are randomized on 32-bit x86. The stack is non-executable by default but can be made executable for legacy applications.

Then, the main executable's image is mapped into memory. GCC generates position dependent executables by default. These may (and often do) contain absolute addresses. However, position dependent executables must be loaded at a fixed address, even if ASLR is enabled. One can also generate Position Independent Executables (PIE), which have been supported on GNU/Linux since 2003. PIE images are loaded at a randomized address and may not contain absolute addresses. Instead, addresses must be computed dynamically, using PC-relative offsets. Because of the extra register pressure that comes with dynamic address computations and because of the limited amount of general-purpose registers on the x86 architecture, GCC still doesn't generate PIE images by default.

Moreover, most Linux distributors will only ship PIE executables for programs which they deem to be security-sensitive. For example, the recently released Ubuntu 14.04 for the AMD64 architecture ships with 1019 programs in its /usr/bin folder, of which only 107 are compiled as PIE executables. Other contemporary distributions ship with a similar number of PIE executables. One may wonder why distributors are putting their users at risk when PIE was proven to have only a marginal impact on performance [92].

If the executable is dynamically linked, the kernel then maps an architecture-specific virtual dynamic shared object (VDSO) into memory. The VDSO may contain specialized code to transfer control from user space to kernel space or specialized versions of commonly used system calls. The VDSO is very small and never spans more than one page of memory (even on AMD64). Its base address is randomized if ASLR is enabled.

If the executable is dynamically linked, the kernel will now map the program interpreter (usually called ld-linux.so.2). The program interpreter will be the first component to be invoked when the kernel transfers control over the program to user space. The interpreter is re-

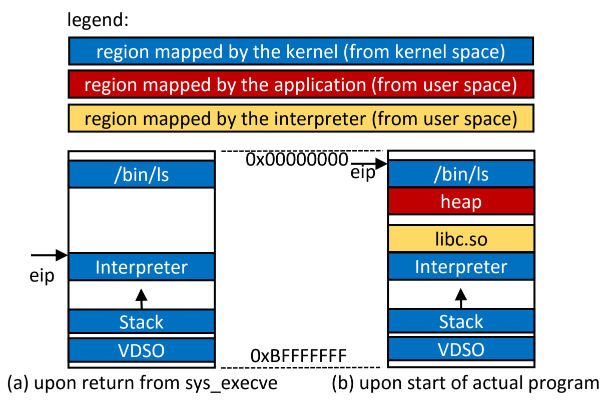

legend:
region mapped by the kernel (from kernel space)
region mapped by the application (from user space)
region mapped by the interpreter (from user space)
0x00000000
eip
/bin/ls
Interpreter
Stack
VDSO
/bin/ls
heap
libc.so
Interpreter
Stack
VDSO
0xBFFFFFFF
(a) upon return from sys_execve
(b) upon start of actual program

Figure 5.1: Address space layout for standard invocation of the ls tool.

sponsible for loading any shared libraries the program may depend on, for performing the necessary load time relocations, and for binding images.

Figure 5.1(a) depicts the process address space layout after return from the sys_execve call. For the sake of completeness, Figure 5.1(b) depicts the layout after the program interpreter has mapped the shared libraries, and after the application itself has allocated its initial heap.

GHUMVEE cannot override the base address of the above components that are mapped directly by the kernel, as there are no RVPs in kernel space. To enable disjoint code layouts for the program image, the program interpreter, and the VDSO, we have to take special measures. Ideally, we want all of these components to be mapped from within user space, where all mapping requests are RVPs, because of which they can be subjected to DCL.

5.1.2 Disjoint Program Images

Mapping the program image from within user space is trivial. It suffices to load a program indirectly, rather than directly, with a slightly altered system call sys_execve("/lib/ld-linux.so.2",{"ld-linux.so.2", "/bin/ls", argv[1], ...}, NULL);

If a program is loaded indirectly, the kernel maps only the program interpreter, the VDSO and the initial stack into memory. The remainder of the loading process is handled by the interpreter, from within user space. Through indirect invocation, GHUMVEE can override the sys_-

mmap2 request in the interpreter that maps the program image.

At this point, it is important to point out that GHUMVEE does not itself launch applications through this altered system call. Instead, GHUMVEE lets the original, just forked-off processes invoke the standard system call, after which GHUMVEE intercepts that system call and overrides its arguments before passing it to the kernel. This way, GHUMVEE can control the layout of the replicae processes it spawns itself, as well as the layout of all the processes subsequently spawned within the replicae. This is an essential feature to provide complete protection in the case of multi-process applications, such as applications that are launched through shell scripts.

5.1.3 Program Interpreter

Even with the above indirect program invocation, we cannot prevent that the kernel itself maps the program interpreter. Hence the indirect invocation does not suffice to ensure that no code regions overlap in the replicae. As mentioned in Section 5.1.1, the interpreter is only mapped when the kernel loads a dynamically linked program.

To prevent that from happening, even when launching dynamically linked programs, we developed a statically linked loader program, hereafter referred to as the MVEE Loader. Whenever an application is launched under the control of GHUMVEE, it is launched by launching the MVEE Loader, and having that loader load the actual application. Launching the MVEE Loader is again done by intercepting the original sys_execve calls in GHUMVEE, and by rewriting their arguments as indicated on the left of the snapshot at time **T0: Startup** at the top of Figure 5.2. In this figure, standard fonts are used for the system calls as invoked by the replicae; bold fonts are used for the rewritten system calls that the GHUMVEE monitor actually passes to the kernel. On the right, snapshots of the address space layouts of the two replicae are shown.

In each replica launched by GHUMVEE, the copy of the MVEE Loader is started under GHUMVEE's control. At the loader's entrypoint, GHUMVEE first checks whether the VDSOs are disjoint. If they are not, GHUMVEE restarts new replicae until a layout as depicted in Figure 5.2 at time **T1: Replica Restart** is obtained. GHUMVEE restarts replicae by waiting until they reach their first system call, which GHUMVEE then changes into a sys_execve call. One minor problem with this approach is that the original sys_execve call cannot simply be restarted. As soon as this call returns, the process image will have been replaced. Consequently, the arguments of the sys_execve call will have been erased from the replica's memory. These arguments include the command-line arguments and environment pointers. GHUMVEE therefore has to find a writable memory page where it can write a copy of the original arguments before the sys_execve can be repeated. Luckily, the interpreter, which was already in the memory when the first sys_execve call returned, is guaranteed to contain a writable page.

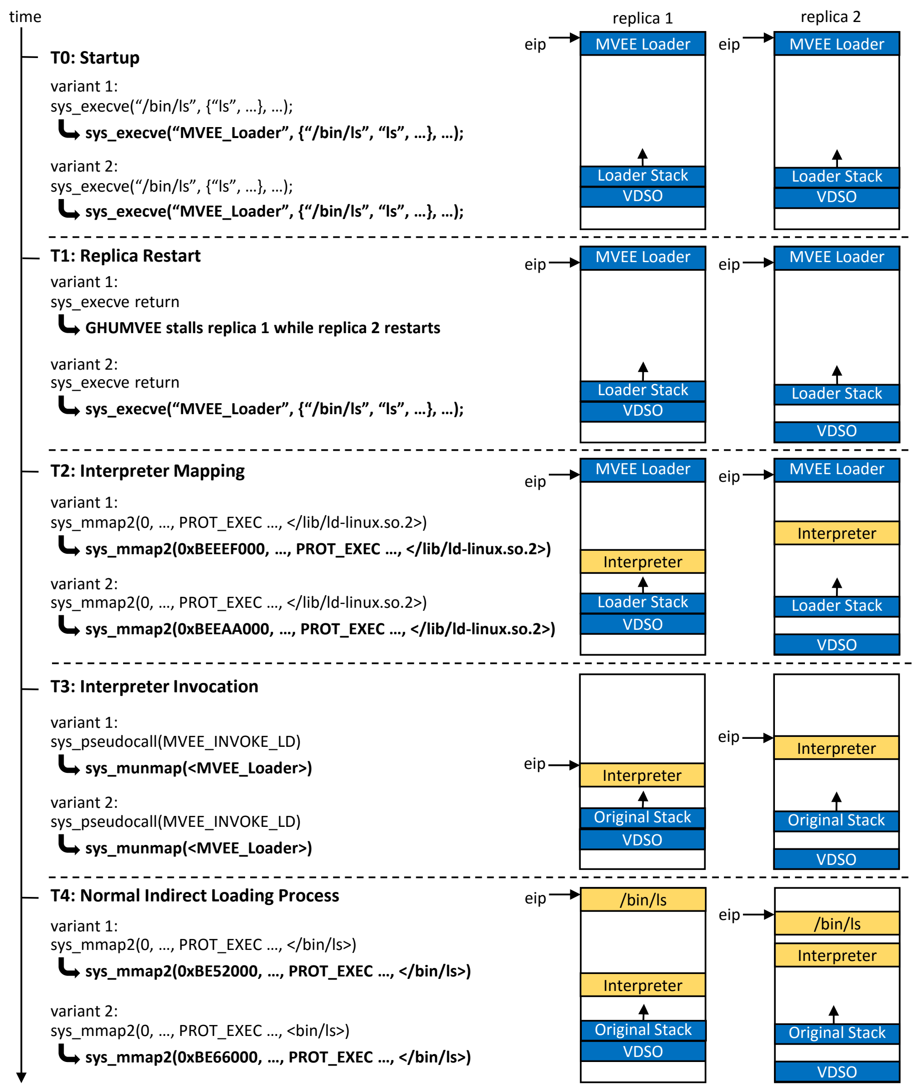
Until recently, the Linux kernel mapped the VDSO anywhere between 1 and 1023 pages below the stack on the i386 platform. It was therefore not uncommon that GHUMVEE had to restart one or more replicae. However, a single restart takes less than 4 ms on our system, so the overall performance overhead is negligible.

After ensuring that the VDSOs are disjoint, the MVEE Loader manually maps the program interpreter through the sys_mmap2 calls shown in Figure 5.2 at time **T2: Interpreter Mapping**. This way, GHUMVEE can override the base addresses of the replicae's interpreters to map them onto regions that contain no code in the other replicae.

Next, the MVEE Loader sets up an initial stack with the exact same layout as when the interpreter would have been loaded by the kernel. Setting up this stack requires several modifications to the stack that the kernel had set up for the MVEE Loader itself. More specifically, we change the first command-line argument from "MVEE_Loader" to "/lib/ld-linux.so.2" and set up the ELF auxiliary vectors that the interpreter would normally get from the kernel [72]. The result is depicted on the right in Figure 5.2 at time **T2**.

The MVEE Loader then transfers control to GHUMVEE through a pseudo-system call, as depicted on the left of Figure 5.2 at time **T3: Interpreter Invocation**. GHUMVEE intercepts this call, and modifies the call number and arguments so that the kernel unmaps the Loader. Upon return from the call to GHUMVEE, it transfers control to the program interpreter. When the replicae resume, they will have the memory layout depicted in Figure 5.2 at time **T3**.

The interpreter will then continue to load and map the original program and the shared libraries, all of which will be subject to DCL, as shown on the left of Figure 5.2 at time **T4: Normal Indirect Loading Process**. Afterwards, the interpreter passes control to the program to end up with the address space layout shown in Figure 5.2 at time **T4**.



T0: Startup

variant 1:
```
sys_execve("/bin/ls", {"ls", ...}, ...);
 ↳ sys_execve("MVEE_Loader", {"/bin/ls", "ls", ...}, ...);
```
variant 2:
```
sys_execve("/bin/ls", {"ls", ...}, ...);
 ↳ sys_execve("MVEE_Loader", {"/bin/ls", "ls", ...}, ...);
```

T1: Replica Restart

variant 1:
```
sys_execve return
 ↳ GHUMVEE stalls replica 1 while replica 2 restarts
```
variant 2:
```
sys_execve return
 ↳ sys_execve("MVEE_Loader", {"/bin/ls", "ls", ...}, ...);
```

T2: Interpreter Mapping

variant 1:
```
sys_mmap2(0, ..., PROT_EXEC ..., </lib/ld-linux.so.2>)
 ↳ sys_mmap2(0xBEEEF000, ..., PROT_EXEC ..., </lib/ld-linux.so.2>)
```
variant 2:
```
sys_mmap2(0, ..., PROT_EXEC ..., </lib/ld-linux.so.2>)
 ↳ sys_mmap2(0xBEEAA000, ..., PROT_EXEC ..., </lib/ld-linux.so.2>)
```

T3: Interpreter Invocation

variant 1:
```
sys_pseudocall(MVEE_INVOKE_LD)
 ↳ sys_munmap(<MVEE_Loader>)
```
variant 2:
```
sys_pseudocall(MVEE_INVOKE_LD)
 ↳ sys_munmap(<MVEE_Loader>)
```

T4: Normal Indirect Loading Process

variant 1:
```
sys_mmap2(0, ..., PROT_EXEC ..., </bin/ls>)
 ↳ sys_mmap2(0xBE52000, ..., PROT_EXEC ..., </bin/ls>)
```
variant 2:
```
sys_mmap2(0, ..., PROT_EXEC ..., <bin/ls>)
 ↳ sys_mmap2(0xBE66000, ..., PROT_EXEC ..., </bin/ls>)
```

Figure 5.2: Address space snapshots during GHUMVEE's DCL program launching.

Assuming that the original program stack is protected by W⊕X, this is rather complicated, but from the user's perspective this completely transparent launching process allows us to control, in user space, the exact base address of every region that might contain executable code during the execution of the actual program launched by the user.

The end result are two replicae with completely disjoint code regions, of which any divergence in I/O behavior caused by a ROP attack successfully attacking one replica, will be detected and aborted by the monitor.

5.1.4 Disjoint Code Layout vs. Address Space Partitioning

As mentioned in Section 5.3, Cox et al. and Cavallaro independently proposed to combat memory exploits with essentially identical techniques they called Address Space Partitioning (ASP) [35] and Non-Overlapping Address Spaces [27] respectively. We will refer to both as ASP.

ASP ensures that addresses of program code (and data) are unique to each replica, i.e., that no virtual address is ever valid for more than one replica. ASP does so by effectively dividing the amount of available virtual memory by N, with N the number of replicae running inside the system. We relaxed this requirement. In DCL, only code addresses must be unique among the replicae, but data address can occur in multiple replicae. So for real-life programs, DCL reduces the amount of available virtual memory by a much small fraction than N.

Another significant difference between both the proposed ASP techniques and DCL is that both implementations of ASP require modifications to either the kernel or to the program loader. Cox' N-Variant Systems was fully implemented in kernel space. This way, N-Variant Systems can easily determine where each memory block should be mapped. Cavallaro's ASP implementation requires a patched program loader (ld-linux.so.2) to remap the initial stack and to override future mapping requests. By contrast, GHUMVEE and DCL do not rely on any changes to the standard loader, standard libraries or kernel installed on a system. As such, DCL can much more easily be deployed selectively, i.e., for part of the software stack running on a machine, similar to how PIE is used for selected programs on current Linux distributions as discussed in Section 5.1.1.

Finally, whereas DCL relies on Position Independent Executables

(PIE) [92] to achieve non-overlapping code regions in the replicae, both presented forms of ASP rely on standard, non-PIE ELF binaries, despite the fact that PIE support was added to the GCC/binutils tool chain in 2003, well before ASP was proposed. Those non-PIE binaries cannot be relocated at load time. Enabling ASP is therefore only possible by compiling multiple versions of the same ELF executable, each at a different fixed address. ASP tackles this problem by deploying multiple linker scripts for generating the necessary versions of the executable. Unlike regular ELF executables, PIE executables can be relocated at load time. So our DCL solution requires only one, PIE enabled, version of each executable. This feature can again help towards a wide-spread adoption of DCL.

5.1.5 Compatibility Considerations

Programs that use self-modifying or dynamically compiled, decrypted, or downloaded code may require special treatment when run with DCL. Particularly, GHUMVEE needs to ensure that these programs cannot violate the DCL guarantees. We therefore clarify how GHUMVEE interacts with the program replicae in a number of scenarios.

Changing the protection flags of memory pages that were not initially mapped as executable is not allowed. GHUMVEE keeps track of the initial protection flags for each memory page. If the initial protection flags do not include the PROT_EXEC flag, then the memory page was not subject to DCL at the time it was mapped and GHUMVEE will therefore refuse any requests to make the page executable by returning the EPERM error from the sys_mprotect call that is used to request the change. This will inevitably prevent some JIT engines from working out of the box. However, adapting the JIT engine to restore compatibility is trivial. It suffices to request that any JIT region be executable at the time it is initially mapped.

Changing the protection flags of memory pages that were initially mapped as executable is allowed without restrictions. GHUMVEE will not deny any sys_mprotect requests to change the protection flags of such pages.

Programs that use the infamous "double-mmap method" to generate code that is immediately executable will not work in GHUMVEE. With the double-mmap method, JIT regions are mapped twice, once with read-write access and once with read-execute access [43, 91]. The code is generated by writing into the read-write region and can then be executed from the read-execute region. On Linux, a physical page can only be mapped at two distinct locations with two distinct sets of protection flags through the use of one of the APIs for shared memory. As we discussed in Chapter 2, GHUMVEE does not allow the use of shared memory. Applications that use the double-mmap method would therefore not work. That being said, in this particular case we do not consider our lack of support for bi-directional shared memory as a limitation. Any attacker with sufficient knowledge of such a program's address space layout would be able to write executable code directly, which renders protection mechanisms such as W$\oplus$X useless. This method is therefore nothing short of a recipe for disaster. In practice, we only witnessed this method being used once, in the vtablefactory of LibreOffice.

5.1.6 Protection Effectiveness

We cannot provide a formal proof of the effectiveness of DCL. Informally, we can argue that by intercepting all system calls, GHUMVEE can ensure that not a single region in the virtual memory address space will have its protections set to PROT_EXEC in more than one replica. Furthermore, GHUMVEE's replication ensures that all replicae receive exactly the same input. This is the case for input provided through system calls and through signals.

Combined, these two features ensure that when an attacker passes an absolute address to the application by means of a memory corruption exploit, the code at that address will be executable in no more than one replica. The operating system's memory protection will make the replicae crash as soon as they try to execute code in their non-executable or missing page at the same virtual address.

Finally, we should point out this protection only works against external attacks, i.e., attacks triggered by external inputs that feed addresses to the program. Artificial ROP attacks set up from within a program itself, as is done in the run-time intrusion prevention evaluator (RIPE) [140], will not be prevented, because in such attacks return addresses are computed within the programs themselves. For those return addresses, different values are hence computed within the different replicae, rather than being replicated and intercepted by the replication engine.

5.2 Experimental Evaluation

We evaluated our technique on two machines. The first machine has two Intel Xeon E5-2650L CPUs with 8 physical cores and 20MB L3 cache each. It has 128GB of main memory and runs a 64-bit Ubuntu 14.04 LTS OS with a Linux 3.13.9 kernel. The second machine has an Intel Core i7 870 CPU with 4 physical cores and 8MB L3 cache. It has 32GB of main memory and runs a 32-bit Ubuntu 14.10 OS with a Linux 3.16.7 kernel. On both machines, we disabled hyper-threading and all dynamic frequency and voltage scaling features. Furthermore, we've compiled both kernels with a 1000Hz tick rate to minimize the monitor's latency in reacting to system calls.

5.2.1 Correctness

To evaluate correctness, we have tested GHUMVEE on several interactive desktop programs that build on large graphical user interface environments, including GNOME tools such as gcalctool, KDE tools such as kcalc and LibreOffice. For, e.g, LibreOffice we tested operations such as opening and saving files, editing various types of documents, running the spell checker, etc. We repeated tests in which GHUMVEE spawned between one and four replicae from the same executable, and tests were conducted with and without ASLR enabled. All tests succeeded.

5.2.2 Usability of Interactive & Real-Time Applications

We also checked the usability of interactive and real-time applications. Except for small start-up overheads, no significant usability impact was observed. For example, with two replicae and without hardware support[1], MPlayer was still able to play 720p HD H.264 movies in real time

[1]For using hardware support, MPlayer tries to obtain shared memory pages with read and write permissions from the kernel. As explained in Chapter 2, GHUMVEE can currently not support the potential bi-directional communication through shared memory with such permissions. As explained in our previous work [133], GHUMVEE therefore intercepts the system call and returns an error value to indicate that the allocation requested to the kernel failed [133]. MPlayer then falls back on its software-only version.

without dropping a single frame, and 1080p Full HD H.264 movies at a frame drop rate of approximately 1%. Because none of the dropped frames were keyframes, playback was still fluent, however.

5.2.3 Execution Time Overhead

To evaluate the execution time overhead of GHUMVEE and DCL on compute-intensive applications, we ran each of the SPEC CPU2006 benchmarks 5 times on their reference inputs.[2] From each set of 5 measurements, we eliminated the first one to account for I/O-cache warmup. On the 64-bit machine we've compiled all benchmarks using GCC 4.8.2. On the 32-bit machine we used GCC 4.9.1. All benchmarks were compiled at optimization level -O2 and with the -fno-aggressive-loop-optimizations flag. We did not use the -pie flag for the native benchmarks. Although running with more than 2 replicae does not improve DCL's protection, we have also included the benchmark results for 3 and 4 replicae for the sake of completeness.

As shown in Figures 5.3 and 5.4, the run time overhead of DCL is rather low overall.[3] On our 32-bit machine, the average overhead across all SPEC benchmarks was 8.94%. On our 64-bit machine, which has much larger CPU caches, the average overhead was only 6.37%. That being said, a few benchmarks do stand out in terms of overhead. On i386, we see that 470.lbm performs remarkably worse than on AMD64. We also see several benchmarks that perform much worse than average on both platforms, including 429.mcf, 471.omnetpp, 483.xalancbmk and 450.soplex. For each of these benchmarks though, our observed performance losses correlate very well with the figures in Jaleel's cache sensitivity analysis for SPEC [62].

A second factor that definitely plays its role is PIE itself. While our figures only show the native performance for the original, non-PIE, benchmarks, we did measure the native performance for the PIE version of each benchmark as well. For the most part we did not see significant differences between PIE and non-PIE, except for the 400.perlbench

[2]Not a single SPEC benchmark needed to be patched for running on top of GHUMVEE. One benchmark, 416.gamess, can trigger a false positive intrusion detection in GHUMVEE because it unintentionally prints a small chunk of uninitialized memory to a file. With ASLR, the uninitialized data differs from one replica to another. In GHUMVEE, we whitelisted the responsible system call to prevent the false positive.

[3]The 434.zeusmp benchmark maps a very large code section and therefore does not run with more than 2 replicae on our 32-bit machine.

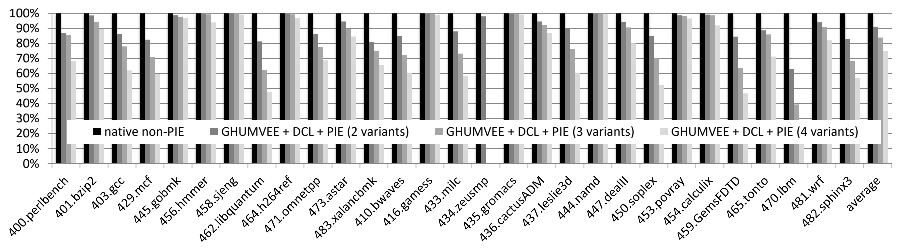


Figure 5.3: Relative performance of 32-bit protected SPEC 2006 benchmarks.

and 429.mcf benchmarks on the AMD64 platform. These benchmarks slow down by 10.98% and 11.93% resp. by simply using PIE.

A final contributor worth mentioning is the system call density. As we discuss at length in Chapter 6, system call processing inside an MVEE can be a major bottleneck. Because of the efficient design of our monitor and because none of the SPEC benchmarks have a high system call density compared to, e.g., the PARSEC benchmarks, this bottleneck is only visible here for benchmarks such as 400.perlbench (362 syscalls/sec) and 403.gcc (1003 syscalls/sec), albeit barely.

5.2.4 Memory Overhead

We examined the memory footprint of our technique on the 32-bit machine. While running benchmarks with 2 replicae, GHUMVEE consumed 9.5MB of physical memory on average. Combined with the duplication of private, writable pages of the first replica, this resulted in a total system-wide memory footprint increase of almost exactly 100%. By comparison, AddressSanitizer increases the memory footprint by 237% on average. Within the replicae themselves, DCL did not introduce direct overhead: Each replica is a separate process that has its full virtual address space available. Each replica maps exactly as much data and code as the native, unprotected programs. Moreover, regions in the address space that may not contain code due to DCL may still be used for data mappings. DCL does, however, introduce some fragmentation, which may marginally reduce the replicae's ability to allocate large blocks.

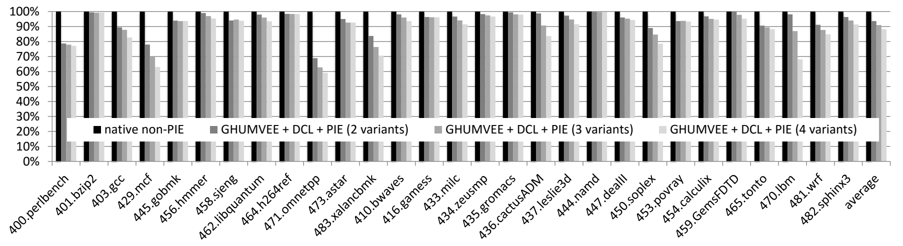


Figure 5.4: Relative performance of 64-bit protected SPEC 2006 benchmarks.

5.2.5 Effectiveness of the Protection

To validate the effectiveness of DCL itself, we constructed four ROP attacks against high-profile targets. The attacks are available at `http://www.elis.ugent.be/~svolckae`.

Our first attack is based on the Braille tool by Bittau et al. [19]. It exploits a stack buffer overflow vulnerability (CVE-2013-2028) in the nginx web server. The attack first uses stack reading to leak the stack canary and the return address at the bottom of the vulnerable function's stack frame. From this address, it calculates the base address of the nginx binary and uses prior knowledge of the nginx binary to set up a ROP chain. The ROP program itself grants the attacker a remote shell. We tested this attack by compiling nginx with GCC 4.8 with both PIE and stack canaries enabled. The attack succeeds when nginx is run natively with ASLR enabled and also when nginx is run inside GHUMVEE with only 1 replica. If we run the attack on 2 replicae, however, it fails to leak the stack canary. While attempting to leak the stack canary, at least one replica crashes for every attempt. Whenever a replica crashes, GHUMVEE assumes that the program is under attack and shuts down all other replica in the same logical process. Despite the repeatedly crashing worker processes, the master process manages to restart workers quickly enough to keep the server available throughout the attack.

While GHUMVEE manages to stop this attack, the attack would probably not have worked even without DCL enabled. After all, with more than one replica, the stack-reading step of the attack can only succeed if every replica uses the same value for its stack canary and the same base address for the nginx binary. To prove that DCL does indeed stop ROP attacks, we have therefore constructed three other attacks against programs that do not use stack canaries and for which we read the memory layout directly from the /proc interface, rather than through stack-reading.

Our second attack exploits a stack buffer overflow vulnerability (CVE-2010-4221) in the proftpd ftp server. The attack scans the proftpd binary and the libc library for gadgets required in the ROP chain, and reads the load addresses of proftpd and libc from /proc/pid/maps to determine the absolute addresses of the gadgets. The gadgets are combined in a ROP chain that loads and transfers control to an arbitrary payload. In our proof-of-concept this payload ends with an execve system call used to copy a file. The buffer containing the ROP chain is sent to the application over an unauthenticated FTP connection. The attack succeeds when proftpd is run natively with ASLR enabled and also when run inside GHUMVEE with only 1 replica. When run with 2 replicae, GHUMVEE detects that one replica crashes while the other attempts to perform a sys_execve call. GHUMVEE therefore assumes that an attack is in progress and it shuts down all replicae in the same logical process. During the attack, proftpd's master process managed to restart worker processes quickly enough to keep the server available throughout the attack.

Our third attack exploits a stack-based buffer overflow vulnerability (CVE-2012-4409) in mcrypt, an encryption program that was intended as a replacement for crypt. The attack loads addresses of the mcrypt binary and the libc library from the /proc interface to construct a ROP chain, which is sent to the mcrypt application over a pipe. The attack succeeds when mcrypt is run natively with ASLR enabled and also when run inside GHUMVEE with only 1 replica. When run with 2 replicae, GHUMVEE detects a crash in one replica and an attempt to perform a system call in the other. It therefore shuts down the program to prevent any damage to the system.

Our fourth attack exploits a stack-based buffer overflow vulnerability (CVE-2014-0749) in the TORQUE resource manager server. The attack reads the load address of the pbs_server process, constructs a ROP chain to load and execute an arbitrary payload from found gadgets, and sends it to the server over an unauthenticated network connection. The attack succeeds when TORQUE is run natively with ASLR enabled and also when run inside GHUMVEE with only 1 replica. When run with 2 replicae, GHUMVEE detects a crash in one replica and an attempt to perform a system call in the other. It therefore shuts down the program to prevent any damage to the system.

5.3 Related Work

We refer the reader to the excellent overview presented by Szekeres et al. for an extensive discussion of existing attacks that exploit memory corruption bugs in software written in low-level languages like C or C++ [126]. Szekeres et al. also discuss why all currently existing defenses fail.

In this section, we discuss the existing techniques more briefly, i.e., in so far as needed to compare our own contributions to the state of the art.

5.3.1 Memory Attacks and Defenses

Every modern operating system supports at least Address Space Layout Randomization [102] and W⊕X [103]. Additionally, nearly every modern compiler enables stack overflow protection [34] by default. Over the years, all of these basic mitigations have been bypassed or hacked.

Shortly after it was introduced, ASLR was shown to be vulnerable to both information leakage attacks [44] and brute-force attacks [122]. On 32-bit x86 platforms, it is especially weak because the 12 least significant bits of addresses cannot be randomized due to page alignment and because the 4 most significant bits often do not get randomized to minimize address space fragmentation [80]. Additionally, Bittau et al. recently demonstrated that even on 64-bit platforms, ASLR brute-force attacks are feasible [19].

W⊕X has not been attacked directly. It can however be bypassed easily. Solar Designer demonstrated return-to-libc attacks as early as 1997 [125], long before W⊕X and its predecessor, non-executable stacks [103], were even deployed. Return-to-libc attacks leverage code already present in the target application to seize control of the application without code injection. Return-to-libc attacks were further improved by Nergal to defeat W⊕X as well as ASLR [95].

In 2007, Shacham presented the first Return Oriented Programming (ROP) attacks [121]. In these attacks, an attacker gains control of the call stack to hijack program control flow. He forces the execution of carefully chosen machine instruction sequences, so-called gadgets, from the program's own code or linked library code, each of which typically ends in a return instruction. It was demonstrated that ROP attacks, unlike return-to-libc attacks, can be crafted to perform arbitrary computations, provided that the attacked application is sufficiently large [51]. Return-to-libc attacks, by contrast, are limited to executing entire functions at once. On architectures with variable length instructions, ROP attacks can additionally leverage code that was not intentionally placed into the application by the compiler, e.g., by transferring control into the middle of instruction encodings as generated by the compiler [121].

ROP attacks were later generalized to other architectures such as SPARC [24], ARM [65], and many others. Despite the progress and activity on the attacker front, defense against ROP attacks is still very much an open problem, even though several solutions have been proposed.

5.3.2 Custom Code Analysis and Code Generation

Dynamic instrumentation tools such as DROP [29] and ROPdefender [39] instrument the protected program at run time to detect ROP attacks. Both tools intercept return instructions and verify the stack before allowing the program to continue. DROP's stack verification consists of calculating the length of the function the program is about to return to and calculating the amount of possible ROP gadgets on the stack. ROPdefender maintains a shadow stack to detect whether or not return addresses are being overwritten. These tools do not require recompilation of the protected program but they slow down the program with factors of 5.3 (DROP) and 2.1 (ROPdefender) on average.

TaintCheck does not specifically target ROP attacks, but its dynamic taint analysis can protect against them and against a wide array of other exploits [96]. TaintCheck does however suffer from large runtime overhead up to 2500%.

Other tools based on dynamic binary translation rewrite a program completely. Hiser et al. [50] proposed Instruction Location Randomization (ILR), a technique implemented in the Strata VM [117, 118]. ILR individually randomizes the location of every instruction within the program and can perform re-randomization at run time. ILR achieves average performance overhead of just 13-16% on the SPEC 2006 benchmarks. It does, however, require an offline static analysis before running a protected program.

Just recently, we've seen two promising tools that target ROP attacks directly. kBouncer and ROPecker both leverage the Last Branch

Recording (LBR) facilities found in recent Intel CPU's [31,59,101] to detect suspicious control-flow patterns. LBR keeps track of the most recently executed branch instructions and their targets. This mechanism allows the tools to identify chains of indirect branches to short gadgets, which are often indicative of an ongoing ROP attack. While these tools hold up quite well in terms of performance overhead and detection of publicly available exploits, there are some fundamental issues with this technique. First, LBR keeps track of a very limited set of branches. In its earliest implementation, only the 4 last branches were recorded. In recent Intel CPUs, up to 16 branches get recorded. Second, when assessing the integrity of the LBR history, it is hard to tell whether or not a branch target might be a ROP gadget and whether or not enough gadgets have been chained together to raise suspicion. As such, these tools would need to be tweaked on a per-application basis to maximize protection while minimizing false positive detections. Göktaş et al. provided more insight into the extent of this problem. They also presented two exploits that bypass both tools [48].

Other compilers attempt to immunize programs against ROP attacks by generating gadget-free code. Li et al. adapted their x86 LLVM compiler to compile "return-free" code [71]. Their compiler never emits any of the x86 return instructions, not even as a part of a multi-byte opcode or instruction operand. They built a custom FreeBSD kernel that was no more than 17.32% slower than the stock kernel. Shortly thereafter though, Checkoway et al. presented a Turing-complete set of ROP gadgets that does not rely on return instructions [28,51].

Onarlioglu et al. presented a similar but more promising technique: G-Free [98]. Through extensive use of alignment sleds, G-Free removes unaligned free branch instructions from a program. Additionally, it protects the remaining aligned free branches to prevent them from being misused. The resulting binaries contain almost no gadgets. G-Free essentially de-generalizes the threat of ROP-attacks to that of less powerful return-to-libc attacks. Onarlioglu et al. report only 3.1% slowdown and a 25.9% increase in binary size on average. It is however doubtful if such performance numbers would hold if G-Free was more extensively evaluated. Only a handful of (rather small) programs were tested with a fully immunized software stack (i.e., with every library compiled using G-Free).

By comparison, Jackson et al. [61] reported higher overhead for their diversifying GCC and LLVM compilers. Similar to G-Free, their compiler adds alignment sleds in front of candidate gadgets in order to remove unintended gadgets from the binary. Unlike G-Free however, their compiler aims to introduce diversity rather than immunity. By adding the alignment sleds only with an arbitrary probability, their compiler can generate many versions of the same program. These versions will have different gadgets, in different locations. The advantage of this approach is that any of the ROP-attacks compiled against one version of a program will only affect a small fraction of the entire user base. If alignment sleds are added with a probability of 1, in which case one would expect the resulting binary to be similar to those generated by G-Free, the overhead on SPECint 2006 benchmarks ranged from 0 to 45%. The authors provide a comprehensive analysis of said overhead and of the effects of NOP-alignment sleds on L1 instruction cache and translation-lookaside buffer (TLB) misses.

Other compiler approaches do not target attacks directly. Instead they focus on enforcing the intended behavior of the program. Stack protectors such as StackGuard insert canaries on the stack to detect overwritten return addresses [34]. LibSafe and many standard C-libraries offer protection against format string vulnerabilities through hardened versions of string functions [8]. Control-flow integrity (CFI) techniques add checks around indirect jumps to detect unintended branch targets [1, 144]. As shown by Göktaş et al. [47], Davi et al. [38] and several others, even the strictest and most fine-grained CFI policies in use do not mitigate ROP attacks completely.

The most recent contribution to this domain is Code-Pointer Integrity (CPI) [66]. With CPI, Kuznetsov et al. isolate all sensitive pointers, which are defined recursively as code pointers and pointers to sensitive pointers. All sensitive pointers are stored in a safe memory that can only be accessed by instructions protected with run-time checks. Thus, guaranteed protection is provided against all attacks that try to exploit memory corruption bugs to hijack control flow by overwriting code pointers. Because relatively few accesses to the sensitive pointers occur, the execution time overhead is limited to around 10% on average. An alternative, more relaxed form of the protection, in which only code pointers themselves are considered sensitive, provides practical protection against all studied existing attacks, at an average cost of less than 2%. As this technique is very recent, no independent validation is available yet. So far, two major potential issues have been raised. First, on some programs, the execution time overhead of CPI turns out to be over 75%. Secondly, in order to identify a conservative overestimation of the set of sensitive pointers, a static data flow analysis is needed, e.g., to handle conversions from pointers to int or long variables and back. That analysis, like all data flow analyses, suffers from aliasing [106]. While Kuznetsov et al. provide an intraprocedural analysis that apparently handles local conversions to int and back pretty well, conversions to void * lead to large overestimations of the set of sensitive pointers, and hence to larger slowdowns. Also, it is at this point unclear whether their intraprocedural analysis (with some interprocedural extensions) suffices to guarantee protection in all cases, incl. legacy or obfuscated code that might not adhere to some of the more recent pointer conversion restrictions in C. Finally, on AMD64 platforms, the protection is not guaranteed (without changes to the OS) because of those platforms' lack of segmentation to isolate the safe memory from the standard memory.

Perhaps the most interesting compiler tool is AddressSanitizer (ASan) [120]. ASan is a memory error checker that, unlike many other memory checkers, instruments the protected program at compile time. ASan instruments all loads and stores and detects a wide array of memory errors. Among these are heap, stack and global buffer overflows. Functionality-wise, ASan is extremely suited to detect and prevent the memory corruption exploits at the basis of ROP and return-to-libc attacks. However, ASan comes with high overhead compared to some of the techniques that target these attacks specifically. The current implementation incurs 73% execution time overhead on the SPEC 2006 benchmarks, as well as 237% memory footprint overhead.

Not to depend on the availability of source code, Pappas et al. proposed to diversify software post compile time [100]. Using in-place code randomization, they demonstrated effective protection against existing ROP exploits and ROP code generators on third-party applications. However, as their technique only provides probabilistic protection rather than complete immunity, it is unclear whether it is future-proof. Moreover, it is unclear whether their static rewriting of binary code is conservative when applied to code that features atypical indirect control flow, such as heavily obfuscated code. In that regard, it is not promising that other recent post compile time binary rewriters, such as SecondWrite [99] and REINS [138], are also explicitly limited to non-obfuscated code.

5.3.3 Replication and Diversification Defenses

Monitoring-based tools leverage kernel or system APIs (application programming interfaces) to monitor program behavior. One important class of monitoring tools are the N-Variant Systems [35] and the conceptually similar Multi-Variant Execution Environments [27, 114, 133]. N-Variant systems run multiple versions (also referred to as variants or replicae) of the same program in parallel. A monitoring component feeds all replicae the same input and then monitors the replicae's behavior. Since all replicae are required to be I/O-equivalent, any differences in behavior trigger an alarm. N-Variant systems have been used to defend against several types of attacks.

The strength of N-Variant systems lies in the fact that each replica can be diversified, as long as the I/O-behavior remains unchanged. By deploying different diversification techniques to each replica, a wide range of attacks can be made asymmetrical, in the sense that they may be able to compromise one replica, but not the other. To cause harm to the system under attack, the successfully compromised replica has to invoke malicious I/O operations that are not part of the intended behavior of the original program, and that will hence not be invoked by the other replica. By synchronizing all I/O operations in all replica, by checking the equivalence of all I/O operations before they are passed to the kernel, and by halting the program when the I/O operations diverge, the monitor can then interrupt any attack before it causes harm.

Salamat et al. demonstrated an N-Variant system that runs replicae with stacks growing in opposite directions [112]. These replicae are generated with a modified version of GCC, with the replicae with stacks growing upwards being only marginally slower. This technique stops even the most advanced stack smashing attacks that do successfully bypass other stack protectors [4].

Salamat et al. also proposed to renumber system calls [113]. At compile time, replicae are generated that each use randomly permutated system call numbers. The monitoring agent dynamically remaps each system call to its original number using the ptrace API, this preventing hackers from injecting code that uses inline system calls. Their use of the ptrace API is similar to how our prototype intervenes in system calls. We refer to Chapter 2 for an extensive discussion on the ptrace API.

Cox et al. [35] and Cavallaro [27] proposed different forms of Address Space Partitioning (ASP). By partitioning the address space and giving one partition to each replica, they ensure that all addresses at which program code or data are stored in a replica, are unique to that replica. So any attack involving an absolute code or data address, such as a libc function entry point or return address, will result in asymmetric and hence detectable replica behavior.

Cox et al. also proposed instruction set tagging as a defense mechanism against code injection [35]. In an offline step, a binary rewriter [129] prepends a replica-specific tag before each instruction. At run time, a dynamic binary translator checks whether or not each instruction is tagged with the appropriate tag [117]. If not, an alarm is raised and execution halts. While this technique was effective at the time of publication, it has been rendered void by the adaptation of W$\oplus$X.

5.4 Conclusions

In this chapter, we presented Disjoint Code Layouts (DCL). When combined with W$\oplus$X and our Multi-Variant Execution environment GHUMVEE, DCL provides full immunity against most memory exploits, including Return Oriented Programming. Unlike other solutions, our technique incurs only a limited execution time overhead of 6.37% on our 64-bit machine and 8.94% on our 32-bit machine. Moreover, DCL does not require a modified compiler or operating system support. Furthermore, programs usually require no or only trivial modifications to enable GHUMVEE-compatibility.

Combined, these features of GHUMVEE make multi-variant execution much more convenient to deploy than the pre-existing state of the art.

Chapter 6

Monitoring Relaxation

Security-oriented MVEEs are traditionally implemented using cross-process (CP) monitors as shown in Figure 6.1(a). Most interactions between the trusted monitor and the untrusted replicae are due to system call invocations. In CP designs, these interactions are costly because they require context switches accompanied by TLB and cache flushes.

One way to reduce the context switching overhead is to move the monitor inside each of the variants. Hosek and Cadar evaluated an in-process (IP) monitor design in which a master variant runs ahead of and replicates program inputs to a set of slaves. This design is depicted in Figure 6.1(b). While this is a very efficient design for reliability-focused MVEEs, the master/slave model with a master running ahead provides far less security than the lock-step execution model of security-focused MVEEs.

In this chapter, we propose a hybrid MVEE design—*ReMon*—shown in Figure 6.1(c). ReMon relies on GHUMVEE, our own CP monitor to enforce lock-step execution for all sensitive system calls and isolate the trusted monitoring logic from the untrusted variants. To increase efficiency, we augment traditional CP monitor designs with a small trusted in-process monitoring (IP-MON) that enables efficient replication of data among variants without context switching.

The traditional security policy of monitoring *all* system calls is overly conservative. Many system calls simply query the system and cannot affect other running processes or the persistent state of the system to cause harm. Only a handful of *sensitive* system calls are likely to be offensively useful. Thanks to the IP-MON component, ReMon supports configurable *relaxation* policies that allow harmless calls to ex-

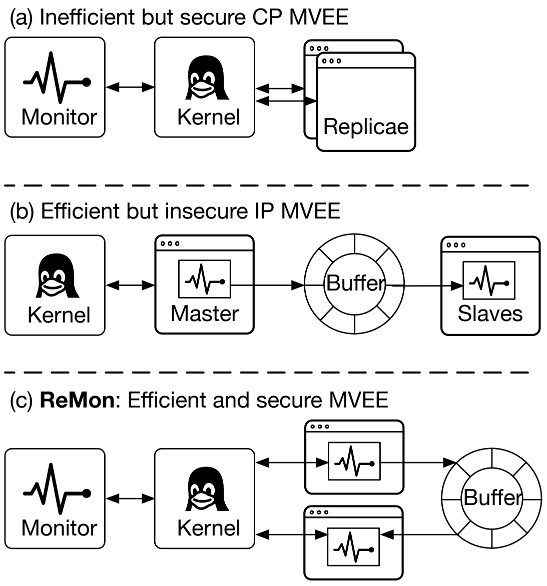

(a) Inefficient but secure CP MVEE
Monitor
Kernel
Replicae
(b) Efficient but insecure IP MVEE
Kernel
Master
Buffer
Slaves
(c) **ReMon**: Efficient and secure MVEE
Monitor
Kernel
Buffer

Figure 6.1: Three MVEE designs running two replicae. The cross-process monitor design (a) is secure but inefficient since interactions between the monitor and the variants require costly context switches. The master/slave design (b) uses an in-process monitor to replicate data to slaves but allows compromise as variants are not executed in lock-step. ReMon (c) combines the security guarantees of cross-process monitoring with the performance of in-process replication.

ecute without being synchronized by our CP-MON component. Section 6.2 evaluates the performance impact of a range of relaxation policies inspired by a recent classification of system calls in the OpenBSD community [105].

6.1 ReMon Design and Implementation

The hybrid MVEE ReMon has three major components:

1. **GHUMVEE**—our security-oriented Cross-Process MVEE.
2. **IP-MON**—an In-Process MVEE implemented as a shared library that resides inside each replica. For a subset of all system calls, IP-MON provides the application with alternative "unmonitored" calls that bypass GHUMVEE. The application interacts with IP-

MON through modified versions of the system libraries, e.g., `glibc`, that call IP-MON instead of the kernel directly.

3. A small **kernel component** which facilitates communication between the other components, in addition to enforcing security restrictions on IP-MON and performing all auxiliary operations that we could not implement in user-space.

This section discusses the design and implementation of the components of ReMon, as well as their interactions.

6.1.1 Dispatching System Calls

The kernel authorizes IP-MON to dispatch system calls as "unmonitored" calls, and does not report such calls to GHUMVEE provided that they originate from an authentic IP-MON. To authenticate IP-MON, the kernel checks that the `r12` register contains an expected magic value (we later discuss how this magic value is securely generated and stored), and that the system call is in a whitelist of allowed calls. Any system call invocation that does not meet these requirements is passed to GHUMVEE.

IP-MON only invokes unmonitored system calls in the master replica, and replicates the results of the system call to the slave replicae through the replication buffer (RB) discussed in Section 6.1.3. Instead of invoking the call directly, the slave replicae wait for the results from the master replica to become available in the RB. In some cases, e.g., when the RB is overflowing, IP-MON explicitly dispatches system calls as monitored calls on both the master and the slave replicae.

Adding support for a new system call in IP-MON is generally straightforward. IP-MON offers a set of C macros that the programmer can use to easily describe how IP-MON should handle the logging of the system call arguments and return values, and how IP-MON should dispatch the system call.

IP-MON currently intercepts 67 system calls. Figure 6.2 shows the IP-MON code for the `read` system call, split across four handler functions. Each handler function implements a different step in the interception of a system call, using the C macros provided by IP-MON. The four steps are:

```
/* read(int fd, void * buf, size_t count) */
MAYBE_CHECKED(read) {
    // check whether our current policy allows us to dispatch read
    // calls on this file as unmonitored calls
    return !can_read(ARG1);
}

CALCSIZE(read) {
    // reserve space for 3 register arguments
    COUNTREG(ARG);
    COUNTREG(ARG);
    COUNTREG(ARG);
    // one buffer whose maximum size is in argument 3 of syscall
    COUNTBUFFER(RET, ARG3);
}

PRECALL(read) {
    // compare the system call arguments dispatch this as a call
    // that only the master actually invokes
    CHECKREG(ARG1);
    CHECKPOINTER(ARG2);
    CHECKREG(ARG3);
    return MASTERCALL | MAYBE_BLOCKING(ARG1);
}

POSTCALL(read) {
    // replicate the results
    REPLICATEBUFFER(ARG2, ret);
}
```

Figure 6.2: Intercepting the `read` system call in IP-MON.

MAYBE_CHECKED This function is called first to determine whether the call should be monitored by GHUMVEE. We use the MAYBE_CHECKED handlers to apply the policies in Section 6.1.2.

CALCSIZE Since we log the system call metadata, arguments and return values all in the same buffer, we need to calculate the maximum size this information may occupy prior to dispatching the call. The CALCSIZE handler is called to determine this size. In case it is bigger than the size of the RB, IP-MON marks the call as a monitored call and it does not log call arguments and return values. In case it is bigger than the available portion of the buffer, the master replica will wait for the slave replicae to catch up and then flush the buffer.

PRECALL If IP-MON decides to dispatch the call as an unmonitored call, then it calls the PRECALL handler next. When called in the context of the master replica, this handler function logs the system call arguments into the buffer. The master replica records information for every system call it executes through IP-MON, regardless of whether or not IP-MON reported the call to GHUMVEE. This information includes the system call number, the system call arguments and the system call return values, in addition to a set of boolean flags that indicate whether or not the call was dispatched as a monitored call, whether the call returns immediately, etc. IP-MON logs all of the system call information except the return values before it dispatches the call. If the function is called in the slave replica's context, then IP-MON instead compares the slave's arguments with the master's arguments.

The return value of the PRECALL handler indicates how the call should be dispatched.

POSTCALL After a system call completes, IP-MON calls the POSTCALL handler. In the master replica, this handler copies the system call return values to the RB.

Whenever slave replicae execute a system call, they wait for the associated information to appear in the buffer. When the system call information appears, the replicae's further actions depend on the aforementioned boolean flags. If the call was marked as a monitored call, then the slave replicae also invoke the call as a monitored call, thus also reporting it to GHUMVEE.

If the call was marked as an unmonitored call, the slave replicae compare their system call number and arguments with those logged by the master replica. If the number or arguments do not match, then the slave replicae trigger an intentional crash, thereby signaling a discrepancy to GHUMVEE. If the number and arguments do match, the slave replicae wait for the results to become available. They either do this in a spin-wait loop (if the master expected the call to return immediately), or by waiting for a specialized condition variable, whose implementation we will describe in Section 6.1.6.

6.1.2 System Call Monitoring Policies

There are many ways to draw the line between which system calls should be monitored and which system calls may be handled by IP-MON. Here, we propose two concrete monitoring exemption policies that draw such a line.

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 6.1: Monitor levels for spatial system call relaxation.

The first option is **spatial exemption**, whereby certain system calls are either unconditionally handled by IP-MON and not monitored by the CP MVEE, and other system calls are handled by IP-MON if their system call arguments meet certain requirements. We propose several predefined levels of spatial exemption in Table 6.1, which the program developer or administrator can choose from. Selecting a level enables unmonitored system calls for all calls in that level, as well as all preceding levels. This provides a linear trade-off between performance and security, with later levels having lower overhead but being potentially less secure.

We picked these system calls so we could provide significant security and preserve correctness, while also improving performance significantly. System calls that relate to allocation and management of process resources and threads, as well as signal handling, are always monitored by the CP monitor. This includes syscalls that (i) allocate, manage and close FDs, (ii) map, manage and unmap memory regions, (iii) create, control and kill threads and processes and (iv) all signal handling system calls. We distributed all remaining system calls over the levels, to allow the programmer/administrator to choose the appropriate balance between performance and security.

The second option is **temporal relaxation**, whereby IP-MON probabilistically exempts system calls from the monitoring policy for a short duration if they are repeatedly approved by the monitor. The reasoning here is that many programs, especially those with high system call frequencies, often repeatedly invoke the same sequence of system calls. If a series of these sequences of system calls gets approved by the GHUMVEE, then one possible temporal relaxation policy is to randomly exempt half of the following identical sequences within some time window or range. Note that deterministic temporal relaxation policies (e.g. "Exempt system calls X, Y, Z from monitoring after N approvals within a M millisecond time window") are inadvisable. Assuming that the temporal relaxation policy is known to adversaries, they will exploit that knowledge to drive the MVEE into a state where potentially dangerous system calls becomes except from monitoring.

6.1.3 The IP-MON Replication Buffer

IP-MON must be embedded into all the replicae. It hence consists of multiple independent copies, one per replica. These copies must cooperate and therefore require an efficient communication channel. Although a socket or FIFO could be used, we opted for a RB stored in a memory segment shared between all replicae.

Unlike other high-performance MVEEs [53], we do not use a circular buffer. When our RB overflows, we signal GHUMVEE using a system call. GHUMVEE intercepts it, waits for all replicae to synchronize, resets the buffer to its initial state, and resumes the replicae. Involving GHUMVEE as an arbiter avoids costly read-write sharing on the position variables, which would hinder multi-threading scalability. Instead, each replica thread only reads and writes its own position.

We rely on memory secrecy to protect the RB from tampering. In a secure implementation, no pointer to the RB should ever be visible to the replicae, except when IP-MON itself is executing. IP-MON should temporarily be allowed to store a pointer to the RB in a processor reg-

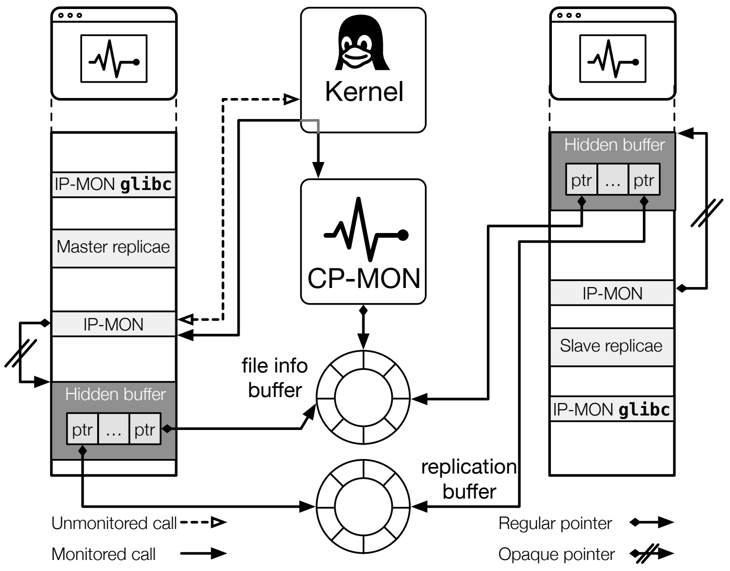


Figure 6.3: Accessing the IP-MON RB through the hidden buffer.

ister, but this pointer should never leak when IP-MON returns the control flow to the program. To achieve this level of security, we use a less commonly used feature of the x86 architecture: segmentation. The kernel allocates a memory region and use its address as the base of the `gs` segment, which IP-MON can then use to access the hidden memory region without knowing its location. The segment base may only be modified from privileged mode, i.e., not user-space[1], so replicae cannot modify it directly. This mechanism therefore allows IP-MON to access a buffer without knowing where it is located in the virtual address space, hiding the buffer from both the library and attackers. Other security work similarly uses segmentation [7,66,90].

However, we sometimes want a temporary pointer reference to the RB. If we have a pointer to the RB, we can use this pointer as the source or destination of optimized `memcpy` and `memcmp` routines. Without the pointer, we are forced to use unoptimized versions `memcpy` and `memcmp` that copy/compare a memory word at a time, since the optimized SSE/SSE2/SSSE3 versions use instructions that do not accept `gs`-relative source or destination operands.

[1]Recent versions of the Intel architecture provide the `RDGSBASE` and `WRGSBASE` instructions that allow user-space to access the segment base, but we can easily disable these instructions with a kernel patch.

Figure 6.3 illustrates how we handle this with an extra layer of indirection. We maintain a single memory page called the *hidden buffer array* to store pointers to buffers that we want to hide from the program. We set the `gs` segment base to the hidden buffer array. When the RB is initialized, we store its address at a known location in this hidden buffer array.

Upon entry to the IP-MON, it moves the RB pointer from its original location to a temporary location in the hidden buffer array. This serves as a security feature: All functions down its callee chain use the temporary location, and will hence malfunction when invoked out of context. Upon exit from IP-MON, we move the pointer back to its original location.

We manually crafted a set of specialized `memcmp` and `memcpy` functions that can accept either their source or destination operands as locations in the IP-MON RB. These specialized functions load the pointer to the RB from the temporary location within the hidden buffer array into a fixed register, and we make sure that the value of the pointer never leaks from these functions.

Our implementation has three clear benefits. First, most of IP-MON's code can refer to any location within the RB using only an offset, rather than a pointer. This minimizes the risk of ever leaking an actual pointer. Second, the manually crafted `memcmp` and `memcpy` routines are small and easy to audit. We have manually verified that our manually crafted routines never leak a valid pointer into the RB to the memory or to any register other than the designated one. Third, our routines cannot be used outside IP-MON since the location from which they load the RB's base pointer only contains said pointer if IP-MON was invoked through its intended entry point.

6.1.4 IP-MON Initialization

IP-MON must be registered with the kernel before it acquires any privileges. We added a new system call that performs this registration. When this call is invoked, the kernel first attempts to report the call to GHUMVEE, which receives the notification and can decide whether or not it wants to allow IP-MON to register.

As a security measure, IP-MON is required to register a whitelist of system calls. When IP-MON later invokes an unmonitored system call, the kernel checks if the call is on the whitelist. If the check succeeds, the kernel allows the call to proceed. Otherwise, IP-MON passes the call to GHUMVEE as a monitored call (alternatively, it could terminate execution and report a security problem). The whitelist is fixed during initialization, and cannot be changed at a later time. However, GHUMVEE can inspect the whitelist and modify this set if needed.

If GHUMVEE approves the registration call, the kernel stores the whitelist in a process-local structure. To support later verification that calls indeed originate from IP-MON, the kernel also stores the base address and size of the IP-MON's executable region (every IP-MON-related call is checked against this region). Finally, the kernel generates a random 32-bit magic key, which is then returned as the return value of the registration call.

GHUMVEE intercepts the return of the registration system call and saves a copy of the magic key, setting the original to zero to prevent the replicae from reading the key directly. When control finally returns to IP-MON, it completes its initialization by mapping the RB through the System V shared memory API (`sys_shmat`). Since no communication channel exists between the replicae at this point and the same RB needs to be mapped in all replicae identically, GHUMVEE allocates the RB itself and overrides the arguments of the `sys_shmat` call to force all replicae to map the same buffer. When the RB is mapped into the replicae's address spaces, GHUMVEE writes the magic key to a known location in the RB, which is only accessible to IP-MON and not to any replica directly.

6.1.5 The IP-MON File Map

GHUMVEE arbitrates all system calls that create/modify/destroy FDs, incl. sockets. This allows it to maintain file meta-data such as the type of each FD (regular/pipe/socket/poll-fd/special). It also keeps track of whether or not a specific FD is in non-blocking mode. System calls that operate on non-blocking FDs return immediately, regardless of whether or not the corresponding operation succeeds.

Replicae can map a read-only copy of this meta-data into their address spaces using the same mechanism we use for the IP-MON RB itself. We refer to this meta-data as the IP-MON File Map. The map can only be accessed through the hidden buffer array in the `gs` segment. We maintain exactly one byte of meta-data for each FD, resulting in a page-sized file map. IP-MON also uses the file map to determine whether some calls are monitored or not (as per the monitoring policies).

6.1.6 Blocking System Calls

The IP-MON file map permits us to predict whether or not a call we dispatch through IP-MON can block or not. IP-MON handles blocking calls efficiently. If it knows up front that a call will block, the master replica instructs the slave replicae to wait on an optimized and highly scalable IP-MON condition variable until the results become available, as opposed to a slower spin-read loop. It uses the `futex (7)` API to implement wait and wake operations and allows us to implement the following optimizations.

For each system call invocation, we allocate a separate structure within the RB. Within this structure, we reserve room for the condition variable. Slave replicae must only wait on the condition variable associated with the system call results they are interested in, greatly increasing the scalability of IP-MON.

We keep track of whether or not there are replicae waiting for the results of a specific system call invocation. This allows us to optimize the case in which no such replicae exist. In this case, the master replica does not need to invoke a `FUTEX_WAKE` operation to resume the slave replicae.

We do not reuse condition variables. Since each system call invocation has an individual condition variable, we do not have to reset them to their initial state after the replicae have been signaled.

6.1.7 Consistent Signal Delivery

Signal handlers may introduce inconsistencies in the execution of a replica. MVEEs therefore typically defer the delivery of signals until they can assert that all replicae are in equivalent states, such as when they are all waiting to enter a system call.

Many intricacies of the `ptrace` API make the implementation of consistent asynchronous signal delivery exceedingly difficult to get right, and this only becomes more complicated when introducing IP-MON. Since GHUMVEE does not see any system calls that are dispatched as unmonitored calls, it might indefinitely defer the delivery of any incoming signals, thus violating the intended behavior of the replicae.

GHUMVEE uses introspection to solve this problem. When a signal is delivered to the master replica, GHUMVEE first sets a *signals pending* flag in that replica's hidden buffer array. Next, GHUMVEE checks whether that replica was executing a system call through IP-MON. GHUMVEE does this by checking whether the user-space instruction pointer points to a system call instruction inside the IP-MON executable region. If the master replica was executing a blocking system call, GHUMVEE aborts that call. The kernel automatically aborts blocking system calls, but normally restarts them after the signal handler has been invoked.

However, GHUMVEE prevents the kernel from restarting the call. Instead, it resumes the master replica at the return site of the call. The master replica then inspects the *signals pending* flag and then restarts the call as a monitored call, allowing it to be intercepted by GHUMVEE.

6.1.8 Support for `epoll (7)`

The `epoll` API is a Linux-specific interface[2] that applications can use to get notifications for FD events, e.g., when a socket has received new data or when a connection request has arrived. Modern Linux server applications use `epoll` to handle network requests efficiently with multiple threads.

To minimize the performance overhead, IP-MON needs to intercept the `epoll` family of system calls. Supporting `epoll` is not straightforward, however. When registering a FD with `epoll` functions, the application can associate an `epoll_event` structure with that FD. This structure may contain a pointer value that the kernel will return when an event on the FD gets triggered. The `epoll_event` structures are a problem for our approach. Diversified replicae are likely to use different pointer values for the same logical FD. Blindly replicating the results of a `sys_epoll_wait` event would then return the master's, rather than the calling replica's pointer values.

IP-MON solves this problem by intercepting all `epoll` system calls, and keeping a shadow mapping between FDs and pointers inside `epoll_event`. When a new FD is registered with `epoll`, we copy the associated pointer value from the `epoll_event` structure to the

[2]Linux 2.5.44 introduced `epoll` as a high-performance alternative to `select` and `poll`.

mapping. When replicating the results of an `epoll` call, we use this mapping to store FDs, rather than pointer values in the master replica, and we map these FDs back onto the associated pointer values in the slave replicae. The shadow mapping is currently implemented as a fixed size array with one pointer per FD, and this implementation has proven sufficient for our experiments.

6.2 Performance Evaluation

We first evaluate the performance of IP-MON's spatial exemption policy in a set of industry standard benchmark suites. We then compare IP-MON with existing MVEEs by replicating some of the experiments previously described in the literature [52,53,85,114,130]. We conducted all of our experiments on a machine with two eight-core Intel Xeon E5-2650L CPUs (20MB L3 cache each), 128GB of RAM and a Gigabit Ethernet connection, running the x86_64 version of Ubuntu 14.04.2 LTS. We used the Linux 3.13.11 kernel, to which we applied the IP-MON patches we described earlier. We used the official 2.19 versions of GNU's `glibc` and `libpthreads` in all of our experiments, and patched them to redirect all unmonitored calls to IP-MON. To maximize the reproducibility of our results, we disabled hyper-threading and all forms of dynamic frequency and voltage scaling in our experiments.

We enabled Address Space Layout Randomization in all of our tests and configured GHUMVEE to map IP-MON and its associated buffers at non-overlapping addresses in all replicae.

6.2.1 Synthetic Benchmark Suites

We evaluated IP-MON on the SPEC CPU2006, PARSEC 2.1, SPLASH-2x and Phoronix benchmark suites[3].

We used the largest available input sets for all benchmarks, and ran the multi-threaded benchmarks with four worker threads and used two replicae for all benchmarks. We could not include PARSEC's `canneal` and `x264` benchmarks, nor SPLASH-2x's `cholesky` benchmark in our measurements. `canneal` is based on data-race recovery and therefore inherently incompatible with our MVEE. `x264` spawns a pool of

[3]Cedomir Segulja kindly provided his data race patches for PARSEC and SPLASH [119].

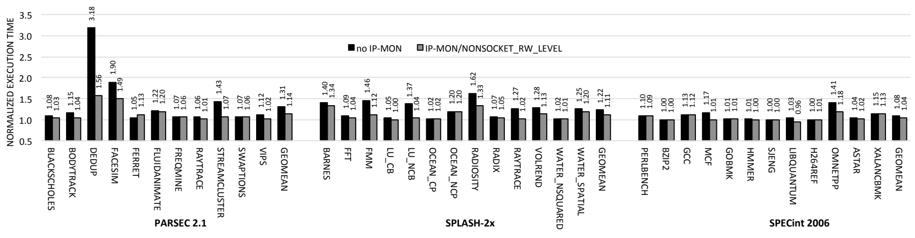


Figure 6.4: Performance overhead for three synthetic benchmark suites (2 replicae)

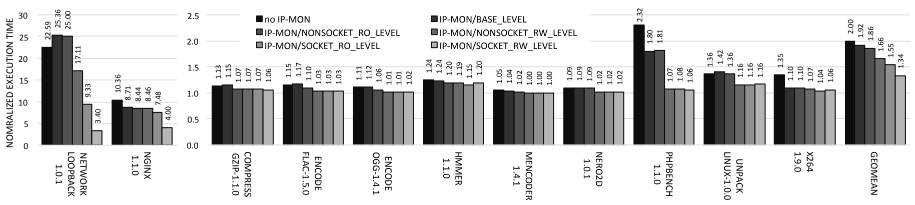


Figure 6.5: Comparison of IP-MON's spatial exemption policies in a set of Phoronix benchmarks (2 replicae)

over 1024 threads, that each map the IP-MON file map and replication buffers, as well as the hidden buffer array. We currently use the System V IPC API to allocate and map all of the aforementioned buffers. Our testing system did not allow us to allocate all of these buffers for this many threads though, because Linux enforces a system-wide limit on the number of allocated System V shared memory segments. We believe that using `mmap`-based shared memory would alleviate this problem. The `cholesky` benchmark finally did not work properly when compiled with the system-provided version of `gcc`.

We ran the Phoronix benchmarks with all five levels of our spatial exemption policy since it was the only suite we tested that contained network benchmarks.

The results for these benchmarks are shown in Figure 6.4 and Figure 6.5. With IP-MON, we see performance gains across all benchmark suites. For SPECint 2006, the relative performance overhead decreases from 8% to 4%. In the PARSEC 2.1 suite, the overhead drops from 31% to 14%. In SPLASH-2x, we see a drop from 22% to 11%. In Phoronix, the overhead drops from 100% to 34%. Particularly interesting are the `dedup` and `network-loopback` benchmarks, in which the overheads drop from 218% to 56% and from 2159% to 240% respectively. These benchmarks invoke system calls at rates of over 60k invocations per second. Furthermore, the Phoronix results clearly show that different policies allow for different security-performance trade-offs.

6.2.2 Server Benchmarks

Server applications are some of the most likely pieces of software to be protected by MVEEs in the future because (i) they are frequently targeted by attackers and (ii) they often run on many-core machines with idle CPU cores that may be used to run parallel replicae.

Not surprisingly, many of the existing MVEEs have been evaluated on server applications. So in this section, we specifically evaluate our MVEE on applications that were already used to evaluate existing MVEEs. These applications include the `Apache` web server (used to evaluate Orchestra [114]), `thttpd (ab)` and `lighttpd (ab)` (used to evaluate Tachyon [85]), `lighttpd (http_load)` (used to evaluate Mx [52]), as well as `beanstalkd`, `lighttpd (wrk)`, `memcached`, `nginx (wrk)` and `redis` (used to evaluate VARAN [53]). We used the exact same client and server configurations as described by the creators of those MVEEs.

We tested IP-MON by running a benchmark client on a separate machine that was connected to our server via a local gigabit link. We evaluated three scenarios. In the first scenario, we used the gigabit link as-is and therefore simulated an (unlikely) worst-case scenario since the latency on the gigabit link was very low (less than 0.5ms). In the second scenario, we added a small amount of latency (bringing the total average latency to 2ms) to the gigabit link to simulate a realistic worst-case scenario — average network latencies in the US are 24–63ms [33]. In the third scenario, which we only evaluated to allow for comparison with existing MVEEs, we simulated a total average latency of 5ms. We used Linux' built-in `netem` driver to simulate the latency [84].

Figure 6.6 shows the worst-case and realistic scenarios side by side. For each benchmark, we measured the overhead IP-MON introduces when running between two and seven replicae side by side with the spatial exemption policy at the `SOCKET_RW_LEVEL`. We also show the overhead for running two replicae with IP-MON disabled. The latter case represents the best-case scenario without IP-MON.

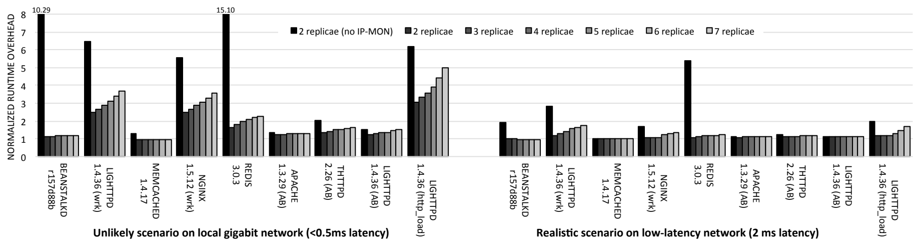


Figure 6.6: Server benchmarks in two network scenarios for 2 up to 7 replicae with IP-MON and 2 replicae without IP-MON.

6.2.3 Comparison with other MVEEs

Table 6.2 compares ReMon's performance with the results reported for other MVEEs in literature [52, 53, 85, 114, 130] and online [132]. As each MVEE is evaluated in different experimental setups, the table also lists some of their features that have a significant impact on the performance overhead. This includes the network latency, because higher latencies hide overhead on the server side, as well as CPU cache sizes, as some of the SPEC benchmarks are memory-intensive and hence their execution, in particular of multiple concurrent replicae, benefits significantly from larger caches.

From a performance overhead perspective, the worst-case setup in which Mx and Tachyon were evaluated had the benchmark client running on the same (localhost) machine as the benchmark server. For VARAN two separate machines reside in the same rack and are hence connected by a very-low-latency gigabit Ethernet.

The worst-case setups in which ReMon and Orchestra were evaluated consist of two separate machines connected by a low-latency gigabit link. In these rather unrealistic scenarios, and with respect to server benchmarks, the differences in setups hence favor ReMon and Orchestra over VARAN, and VARAN over Tachyon and Mx.

In the best-case setups in which Mx and Tachyon were evaluated, one of the machines was located at the US west coast, while the other was located in England (Mx) or the US east coast (Tachyon). In ReMon's best-case setup, we used a gigabit link with a simulated 5 ms latency. So in the more realistic setups and for the server benchmarks, the differences favor Mx and Tachyon over ReMon.

This comparison demonstrates that ReMon outperforms existing security-oriented MVEEs in terms of overhead, and that it at least ri-

Orientation	**Reliability**						**Security**			
MVEE	**Tachyon**			**Mx**		**VARAN**	**Orchestra**	**GHUMVEE**	**ReMon**	
network	local-host	local few hops	coast-to-coast	local-host	USA-UK (150 ms)	same rack gigabit	local gigabit	n/a	local gigabit	local gigabit (5 ms)
CPU cache size				8 MB		8 MB		20 MB	20 MB	
reported overheads										
apache (ab)						**2.4%**	50%		21%	2.5%
lighttpd (ab)	790%	272%	30%			**0.0%**			22%	4.7%
thttpd (ab)	1320%	17%	**0%**			**0.0%**			37%	2.7%
lighttpd (httpld)				249%	4%	**1.0%**			205%	3.0%
redis				1572%	5%	6%			66%	**0.0%**
beanstalkd						52%			15%	**0.6%**
memcached						14%			**0.0%**	**0.0%**
nginx (wrk)						28%			147%	**1.3%**
lighttpd (wrk)						12%			147%	**1.9%**
SPEC CPU2006				17.9%				7.2%	**3.4%**	
SPECint 2006				17.6%		14.2%		12.1%	**4.2%**	
SPECfp 2006				18.3%				3.8%	**2.9%**	

Table 6.2: Comparison with other MVEEs (2 replicae)

vals with the reliability-oriented MVEEs.

6.3 Security Analysis

Unlike previous MVEEs, ReMon eschews fixed monitoring policies and instead allows security/performance trade-offs to be made on a per-application basis.

With respect to integrity, we already pointed out that a CP MVEE monitor (and its environment) are protected by (i) running it in an isolated process space protected by a hardware-enforced boundary to prevent user-space tampering with the monitor from within the replicae; (ii) by enforcing lock-step, consistent, monitored execution of all system calls in all replicae to prevent malicious impact of a single compromised replica on the monitor; (iii) diversity among the replicae to increase the likelihood that attacks cause observable divergence.

With those three properties in place, it is exceedingly hard for an attacker to subvert the monitor and to execute arbitrary system calls. Nevertheless, MVEEs do not protect against attacks that exploit incorrect program logic or leak information though side-channel attacks. This is similar to many other code reuse mitigation techniques such as diversity, SFI and CFI.

In ReMon, monitored system calls are still handled by a CP monitor so malicious monitored calls are as hard to perform as in existing

CP MVEEs. For unmonitored system calls, however, IP-MON provides more leeway by relaxing two of the three properties. As the master replicae can run ahead of the slaves and the system call consistency checks in the slaves' IP-MON, the attacker can try to hijack the master's control with a malicious input to execute at least one, and possible multiple, unmonitored calls without verification by a slave's IP-MON. Because the master's IP-MON is in the master's address-space, the attacker can try to use the hijacked control to hijack the data replication to the slaves. He might exploit this to introduce consistent system call behavior into the slaves (rather than letting the slave(s) crash on the input that successfully hijacked the master) or to ensure that the slaves do not reach their IP-MON for some time, thus letting the master run ahead of the slaves for a longer time, and thus increasing the window of opportunity to execute unmonitored system calls in the master. So in theory, as long as the hijacked master does not execute monitored calls, the attacker can keep executing unmonitored calls.

As long as the attacker only executes unmonitored calls according to a given relaxation policy, those theoretic capabilities pose no significant security treat. However, an attacker might also try to bypass IP-MON's policy verification checks on conditionally allowed system calls to let IP-MON pass calls unmonitored that should have been monitored by GHUMVEE according to the policy. We must therefore consider several aspects of these hypothetical attacks.

Unmonitored execution of system calls Our ReMon kernel modifications ensure that only a registered IP-MON can execute system calls. This prevents any system calls by reusing code (gadgets) outside the IP-MON.

To also prevent an attacker from leveraging IP-MON's elevated privileges to execute unmonitored system calls while bypassing IP-MON's verification checks, we have two security measures in place. First, the kernel guarantees that the initial registration of IP-MON (by the MVEE) and any attempts to manipulate or unmap its executable region are reported to GHUMVEE. This security measure ensures that unmonitored system calls originate from an authentic IP-MON to prevent mimicry attacks. Secondly, the kernel only allows system calls to bypass GHUMVEE if the correct IP-MON key is passed in the `r12` register. This essentially enforces a specialized CFI policy on system calls. First, we only store the key inside IP-MON's RB. Second, we have structured our code such IP-MON only loads the key at its entry point. IP-MON stores the key in a fixed register and it clears this register on any path that leads from the entry point to a return instruction. This way, we ensure that unmonitored calls are always verified by IP-MON and that they can only originate from within IP-MON itself.

The only remaining way to abuse IP-MON's elevated privileges would therefore be a code reuse attack whereby the attacker manually loads the IP-MON key and then directs the control flow to a system call instruction in IP-MON's code. Our handling of this key guarantees that IP-MON never leaks it itself. The attacker would therefore have to locate the key inside the RB. This is difficult, however, as we discuss later.

Manipulating the RB through IP-MON. To prevent an attacker from reusing the master's IP-MON code that fills the RB, we protected this code using a CFI policy similar to the one that protects the IP-MON key: The code that manipulates the buffers can only execute correctly if IP-MON was invoked through its entry point. Once again, the remaining option is to manipulate the IP-MON RB directly.

Locating and accessing the RB directly. As we showed in Section 6.1.3, we hide this buffer's location by accessing it only through the `gs`-relative hidden buffer array indirection. To gain access to the RB, an attacker must therefore either (i) blindly guess the location of the hidden buffer or of the RB or (ii) use gadgets available in the process to read from or write to `gs`-relative memory locations.

The entropy of the address of the 4K page that contains the 1-page hidden buffer array is 36 bits, as $2^{36} - 2$ is the number of user-page pages in the AMD64 ABI's 48-bit canonical address space from which ReMon can choose. This provides much more effective entropy than the implementation of Code Pointer Integrity [66] that relied on hiding a very large (2^{42}) safe area in memory [45]. Furthermore, ReMon ensures that the buffers are located at different addresses in all replicae. The attacker hence needs to find the buffer in all replicae to avoid crashing any one of them.

ReMon's current implementation uses RBs that are 16MiB big and located on different addresses in each replica. This gives 24 bits of entropy per replica.

A remaining alternative for the attacker is to reuse existing code with in the program to obtain direct access to the hidden buffer array. As already mentioned, the `RDGSBASE` and `WRGSBASE` instructions can easily be disabled for monitored processes with a kernel patch. Aligned instructions with `gs`-relative addressing are virtually impossible to find as the `gs` segment is no longer used in AMD64 GNU/Linux user-space software. Finding unaligned gadgets with `gs`-relative addressing by means of a ROP chain that scans the memory in the master is possible in theory, but the time window in which they can be exploited is dramatically reduced by applying a diversification like DCL, because that diversification ensures that the slaves will crash as soon as they get to the injected ROP chain in which the addresses are valid code addresses in the master, but not in the slaves.

Diversified Replicae ReMon can deploy the combined diversification of ASLR, DCL, and stacks growing in opposite directions, i.e, the diversifications previously evaluated in the literature on MVEEs [114, 130]. The security evaluations in that literature, including demonstrations of resilience against real-life attacks, therefore still apply to our ReMon MVEE.

Security Analysis Summary The presence of IP-MON in the replicae's address spaces does open possibilities for attackers to mount asymmetrical attacks or execute arbitrary sequences of unmonitored system calls. The bar for exploitation of IP-MON's infrastructure to mount asymmetrical attacks is extremely high, however. Furthermore, unmonitored system calls remain subject to the kernel's active security policy.

6.4 Related Work

Directly intercepting system calls—known as **system call interposition**—to check whether they are in line with a foreseen system call policy (often obtained through profiling and software analysis) predates MVEEs as a security sandbox technique. The initial literature on the subject identified [49] the high overhead of `ptrace` on Linux (compared to similar techniques on other OSes), and kernel-based implementations were presented to overcome this overhead [104]. To reduce the impact on the kernel, ReMon performs most monitoring IP, and requires only a small kernel patch to ensure its security.

Dune provides **IP but across-privilege-ring monitoring** capabilities based on modern x86 hardware virtualization support such as VT-x and Extended Page Tables (EPT) [14]. This approach could be an interesting alternative to CP-MON's `ptrace`-based activation.

The DieHard MVEE for probabilistic memory safety uses **CP pipe-based replication** [18]. Rather than intervening in I/O system calls, DieHard only intercepts `stdin/stdout` I/O operations.

Cox et al. presented and evaluated an **IP kernel-space MVEE** implementation that deployed address-space partitioning as a diversification technique [35], which can be seen as a limited form of DCL. They measured Apache latency increases of 18% on unsaturated servers, and throughput decreases of 48% on saturated servers, which correspond to much higher overheads than the ones we report for ReMon.

Later **CP user-space** MVEEs, including the one by Cavallero et al. [23], Orchestra by Salamat et al. [114], and GHUMVEE by Volckaert et al. [130, 133] rely on, and suffer from, the properties of the `ptrace` and `waitpid` APIs. They mainly differ from ReMon in the way the replication is performed.

In Orchestra, the monitor executes system calls on behalf of the replicae. To copy large amounts of data between replicae and the monitor efficiently, Orchestra allocates a shared memory buffer between the monitor and the replicae in which both processes can read and write large blocks of data using simple IP `memcpy` operations. GHUMVEE initially relied on two custom `ptrace` extensions to allow the monitor to copy large data blocks directly between replicae. With the introduction of the `process_vm_readv` API in Linux 3.2, these extensions became mostly useless and were abandoned. To allow interception of system calls handled in the VDSO, which are not passed to the OS and consequently not intercepted with `ptrace`, GHUMVEE hides the VDSO such that the replicae fall back on regular system calls that are intercepted.

In addition to its CP system call handling and CP monitoring, GHUMVEE allocates shared buffers between the master and slave replicae to support **IP user-space replication** of synchronization events, incl. pure user-space events such as uncontended futexes, without costly intervention of the CP monitor [133]. GHUMVEE's approach is similar to the replication in Respec [69], the online record-replay system, but includes adaptations to support security-oriented monitoring of diversified replicae that need to execute in lock-step and in which locks reside at different addresses in the different replicae. Likewise, GHUMVEE uses shared buffers, in combination with interposers, for IP replication of address-dependent behavior [133].

VARAN takes this approach one step further, and also performs **IP user-space monitoring** [53] through shared ring buffers as shown in Figure 6.1(b) to avoid the overhead of `ptrace`. In VARAN, the direct master-slave communication is implemented by rewriting the system call instructions (incl. VDSO ones) in the binaries into trampolines to system call replication agents. The agents in the master replica execute the I/O system calls and log them in the shared buffer. The agents in the slave replicae running behind the master then copy the results instead of executing the calls. Monitors embedded in replica processes check the system call consistency, and can even allow small discrepancies between the system calls behavior of the replicae. VARAN does not replicate user-space synchronization events, however, and hence cannot handle many typical client-side applications, most of which rely on user-space futexes these days.

With its support for small system call behavior discrepancies, as well as with some of its design and implementation options to minimize overhead, VARAN positions itself as a reliability-oriented MVEE that can support applications such as transparent failover, multi-revision execution (possibly to detect attacks, but not to prevent them), live sanitization, and record-replay [53]. With its in-process replication avoiding `ptrace`, VARAN significantly outperforms Tachyon [85] and Mx [52], two other reliability-oriented MVEEs.

As already noted by its authors, however, VARAN is less fit to protect against memory exploits. First, VARAN lets the master run ahead of the slaves, even with regards to sensitive system calls, as it does not differentiate between sensitive and insensitive ones in that regard. This leaves a much larger window of opportunity to attackers than ReMon, including for the execution of sensitive calls. Although this window can be shortened by decreasing VARAN's shared ring buffer's sizes to one, it is unclear what the impact on performance will be and whether that buffer adaptation closes the window completely or merely shortens it to one sensitive system call, which would clearly still be too much. Secondly, unlike the many discussed protection techniques implemented for our ReMon's IP-MON, VARAN's IP monitors are only protected from code reusye attacks by ASLR, which has proven susceptible to attacks due to low entropy and granularity [9, 55, 122, 123]. This is all the more problematic as VARAN's IP monitors also monitor the sensitive system calls. Finally, VARAN only rewrites explicit system call instructions in binary code into trampolines to its replication agents. Unlike in ReMon, where all executed system calls from outside the IP-MON are intercepted via the kernel and CP-MON, unaligned system call gadgets in the original binary hence remain unprotected in VARAN, and hence available to ROP attacks.

SFI [57, 86, 136, 141] and CFI [1, 2, 25] are two defenses that have received a lot of attention in literature and that MVEEs can use to protect against memory exploits. Compared to MVEEs such as ReMon, they have the drawback of depending on relatively intrusive code transformations, most of which can only be applied when source code is available, and most of which, in particular the ones with stronger security guarantees, come with a significant performance penalty.

ReMon does not need intrusive transformations in the application's source code. It suffices to have position-independent executables to support DCL, and as we discussed in Section 6.3, DCL applied to an IP-MON that embeds an ad hoc CFI solution can provide quite strong security guarantees, without having to pay a heavy price in execution time increases.

6.5 Conclusions

MVEEs uses the multi-threading capabilities of modern processors to sandbox and monitor software prone to memory corruption and exploitation thereof. Designers of MVEEs face the mutually conflicting goals of security and program performance. Specifically, frequent interactions between cross-process MVEE monitors and the replicae require a high number of costly context switches. We address this challenge through a split-monitor design in which an in-process monitor replicates inputs among the replicae and a cross-process monitor enforces lock-step execution of potentially harmful system calls; innocuous system calls, on the other hand, proceed without external monitoring to increase efficiency.

We present a careful and detailed security analysis and conclude that our introduction of an IP-MON component and relaxed monitoring of innocuous system calls is possible while offering a level of security comparable to that of CP MVEEs. Finally, our extensive performance evaluation shows that the overheads of ReMon ranges from 0-4.7% on a number of realistic server workloads and compares very favorably to previous MVEE designs including recent in-process designs.

Chapter 7

Conclusions and Future Work

In Chapter 3 of this dissertation, I proposed new techniques to extend the principles of MVEEs to programs that rely on implicit inputs. Next, in Chapter 4, I discussed the issue of non-deterministic multi-threaded replicae and presented strategies and replication agents that enable our MVEE to support such replicae. In Chapter 5 I proposed a new, non-probabilistic diversification technique that offers strong protection against memory exploits. Finally, in Chapter 6 I introduced the concept of monitoring relaxation and presented a novel split-monitor design that greatly enhances the performance of our MVEE.

In this chapter, I summarize the conclusions of this dissertation and recommend lines of future work.

7.1 Replication of Implicit Inputs

Program replicae that run inside an MVEE must be fed consistent input in order to guarantee that they will behave the same. Older MVEEs build on the assumption that all program input either originates from the system call interface, or can be stopped at the system call interface by, e.g., disallowing the use of shared memory. As we discussed in Chapter 3 however, this assumption is false.

x86 processors expose (mutable) timing information through unprivileged machine instructions. These instructions can be executed without being supervised by the MVEE's monitor. Similarly, modern versions of Linux use virtual system calls, which are not reported to the MVEE's monitor. These virtual system calls must originate from the Virtual Dynamic Shared Object (VDSO), which is loaded into every running program's address space by the kernel. We proposed to hide this VDSO from the replicae in the MVEE, thereby forcing them to fall back to regular system calls. Additionally, Linux as well as other commodity operating systems allow programs to map shared memory, which allows them to communicate directly and without being supervised by the monitor. Finally, we identified several common programming practices in which implicit program inputs might alter the behavior of the program. Such implicit inputs are likely to differ for each replica as a result of diversification.

In Chapter 3, we proposed workarounds and solutions for all of the above problems. Unprivileged machine instructions that provide mutable input can be disabled using the processor's control registers. Invoking a disabled instruction triggers a protection fault, which the MVEE monitor can intercept to emulate the original instruction and provide the replicae with the expected results.

The use of shared memory can be restricted to prevent the replicae from reading inconsistent input. These restrictions do not prevent programs from executing correctly in practice.

To tackle the reliance on implicit inputs, we proposed to use implicit input replication agents and presented an API and infrastructure that greatly facilitates the implementation of such agents.

7.2 Replication of Parallel Programs

Many parallel programs are non-deterministic by nature and will appear to behave differently from run to run when observed from the system call interface, even if they are not diversified. Scheduling is the root cause of this non-determinism. If parallel programs are allowed to run freely, the order in which they execute instructions that participate in inter-thread communication, or the order in which the effects of these instructions become visible to other threads will change from run to run.

In Chapter 4, we pointed out that Deterministic MultiThreading (DMT) and Record+Replay (R+R) systems seem like a natural fix for this problem. DMT systems impose a deterministic order on inter-thread communication instructions by establishing a fixed schedule for each given program input. As we discussed however, some DMT systems cannot establish such a schedule for programs with threads that perform unbounded computations or indefinitely blocking system calls. Other DMT systems establish a schedule that is tightly bound to program properties that are likely to change as a result of diversification.

R+R systems do not suffer from the same issues but need to be adapted before they can be used in the context of an MVEE. Specifically, R+R agents need to be address-agnostic, RVP neutral and support ad hoc synchronization.

In Chapter 4, we presented three replication strategies and four replication agents that fit within these constraints. One of our replication agents communicates over a secured channel. Additionally, we recommended practical strategies to embed our replication agents into programs and libraries that use ad hoc synchronization.

Our replication agents make GHUMVEE the first MVEE to support arbitrary multi-threaded replicae with limited effort. The wall-of-clocks replication agents are efficient and scalable. When running the PARSEC benchmark suite with four worker threads and two replicae, the wall-of-clocks agents achieve slowdowns of just 1.32x, which is comparable to the slowdowns reported by authors of older MVEEs for single-threaded benchmarks.

7.3 Disjoint Code Layouts

The same software we wish to protect using our MVEE, is frequently targetted by hackers with code reuse attacks. Such attacks diverge the intended control flow of the target to a (set of) known location(s) so as to perform malicious actions chosen by the attacker in the context of the target program.

Cox [35] and Cavallaro [27] independently proposed combat code reuse attacks by splitting the replicae's address spaces into n partitions, with n the number of concurrently executing replicae, and confines each replica to its own partition.

In Chapter 5, we argued that this approach is impractical, however, and proposed Disjoint Code Layouts (DCL) as a practical alternative. DCL achieves the same protection strength as partitioning, and is also efficient. We report performance overheads as low as 6.37% for the SPEC CPU 2006 benchmark suite running on top of a 64-bit Linux 3.13 OS.

7.4 Monitoring Relaxation

As we pointed out in Chapter 2, most security-oriented MVEEs, including GHUMVEE, run the replicae and the monitoring component as separate processes. This approach has obvious security benefits as compromised replicae cannot attack the monitor directly. The cross-process monitor does significantly degrade the performance of the replicae, however. The in-process monitor used in VARAN yields better performance but does not provide the same security benefits as the security-oriented MVEEs [53].

In Chapter 6, we propose a hybrid design for a security-oriented MVEE. Our proof-of-concept implementation, ReMon, combines GHUMVEE with an in-process monitor, IP-MON. ReMon relaxes GHUMVEE's monitoring policy by offloading the handling of innocuous system calls to IP-MON. We proposed and evaluated several relaxed policies and showed that even our most relaxed policy yields similar security guarantees to GHUMVEE, whilst achieving much better performance.

7.5 Future Work

7.5.1 Compiler-based Transformation of Problematic Code

Throughout this dissertation, we have identified several issues that may prevent unpatched replicae from running correctly inside GHUMVEE. In Chapter 3 we discussed address sensitive behavior and presented our implicit input replication API to tackle the problem. In Chapter 4 we discussed programs that rely on ad hoc synchronization or lock-free algorithms, and presented a strategy to patch such programs so that they run correctly in the MVEE.

Tools that could automate the identification and patching of these problematic features would be tremendously useful. Such tools could be built on top of a compiler framework like LLVM[1].

Address-sensitive behavior can be identified by instrumenting all casts between pointer types and integer data types at the IR level. Whenever a pointer is cast to an integer, a call to the implicit-input-replication agent could be inserted such that the integer representation

[1]http://llvm.org/

reflects the master replica's address space layout. Along with the call, the compiler could insert the necessary meta-data such that the result of the reverse cast, i.e., from the integer type back to a pointer type, would reflect the slave replica's memory layout.

This tool could then be optimized to decrease its performance and security impact. Through static techniques such as the compiler's data flow analysis, and dynamic techniques such as taint analysis [96], the instrumentation could be limited to pointers whose integer representation is used in truly address sensitive code. Furthermore, it would be desirable to cast the pointers to a normalized form, rather than to the value determined by the master replica.

Similarly, a tool could be developed to automate the identification and instrumentation of sync ops. As we explained in Chapter 4, sync ops are operations that need to be executed in the same order in all replicae to ensure that they will behave consistently.

A colleague has developed an embryonic LLVM-based tool that identifies sync ops based on the strategy we laid out in Section 4.1.4. This tool translates ad hoc synchronization operations into standardized C++11 atomic operations by modifying the data types for variables on which the operations are performed. These standardized operations can be easily instrumented using our automatically generated header file.

There are many ways in which the current tool can be improved, however. First, it needs to be extended such that it can propagate the modified type information to other translation units. Second, it needs to be extended such that it can identify sync ops in inline assembly code. Finally, it needs to be extended to identify and instrument unprotected load and store instructions. These unprotected load and store instructions for the most common form of benign data races [63].

7.5.2 Reducing Context Switching Overhead

In Chapter 6 we discussed the performance impact of Cross-Process MVEEs and pointed out that such MVEEs suffer from high context switching overhead. We proposed an alternative split-monitor MVEE design that relaxes the MVEE's monitoring policy in order to reduce the number of context switches.

An other alternative would be to reduce the overhead on context switches themselves by leveraging the processor's instruction set extensions. Two of these extensions, VT-x and SGX, allow for the monitor to be placed in the same virtual address space as its replicae.

With VT-x, the monitor can run at the hypervisor privilege level and the replicae can interact with the monitor through VMCALL instructions. The monitor's code and data integrity can be safeguarded using extended page tables (EPT) [58]. This approach has been suggested by Belay et al., but has not been implemented in an MVEE yet [14].

With SGX, the monitor can be placed in an enclave and the replicae can interact with the monitor through EENTER instructions. The monitor's code and data integrity would be safeguarded by design since it would be placed in an enclave [60].

7.5.3 Record/Replay

The synchronization replication agents we presented in Chapter 4 improve upon the existing Record/Replay (R+R) systems because they record and replay invocations of low-level atomic operations (sync ops), rather than just the invocations of high-level pthread-synchronization operations.

An interesting line of future work would therefore be to either extend GHUMVEE with offline R+R capabilities, or to embed GHUMVEE's synchronization replication agents into an existing R+R system. This would, in both cases, result in an R+R system that replays the execution of parallel programs more truthfully than existing R+R systems currently do.

7.5.4 Twin Debugging

As we explained in Section 2.4, GHUMVEE contains much of the same functionality one would expect to see in a debugger like GNU's gdb. This functionality makes GHUMVEE an ideal base to implement a fully-featured twin debugger. A twin debugger can debug two versions of the same program at the same time. This can facilitate, e.g., finding the root cause of a failed regression, by comparing the internal state of the two versions throughout their execution. This functionality could also be used to extend a delta debugger [142]. Delta debuggers automate bug testing by finding the code changes that introduced the bug and by simplifying the program input that must be provided to introduce the bug. Twin debugging functionality could speed up this process since many program versions could now run side by side.

7.5.5 Checkpoint/Restore and Transparent Failover

When GHUMVEE detects a divergence, it shuts down the entire process, and its equivalent processes in other replicae, in which the divergence was triggered. An interesting line of future work could be to only shut down the diverging replica. We currently cannot do this for two reasons.

First, if the diverging replica is the master, then a new master must be elected. As we explained in Section 3.3, GHUMVEE currently cannot do this as the master replica is the only replica that opens file descriptors for non-regular files such as sockets. We could solve this problem by transferring the master's open file descriptors to the newly elected master replica over a UNIX domain socket [139]. This method is also used in VARAN [53].

Second, reducing the number of concurrently executing replicae may reduce the reliability and security guarantees of the MVEE. Ideally, the MVEE should spawn a new replica whenever it shuts down a diverging replica. This new replica should, however, be brought into a state that is equivalent to that of the other replicae. The obvious way to do this is to create periodic checkpoints of the replicae and to restore one of the checkpoints in the newly spawned replica. This solution has long been infeasible because it requires intrusive kernel changes. With the advent of Linux's built-in Checkpoint/Restore functionality however[2], this is no longer the case.

7.5.6 TOCTTOU Race Conditions

GHUMVEE validates system call RVPs by copying each replica's register contexts and system call arguments to the monitor's own address space, and subsequently comparing them. If the replicae pass GHUMVEE's validation, they will be resumed by the monitor, and they will be allowed to complete the system call. There is a small time window between the validation and the completion of the system call in which malicious threads in the replicae may overwrite the validated system call arguments with malicious arguments that would not have passed the validation. McPhee defined this type of vulnerability as a

[2]http://criu.org

Time of check to time of use (TOCTTOU) race condition [87]. This is a common problem for all user-space monitoring tools since no API currently exists to prevent system call arguments from being overwritten. However, Kim and Zeldovich recently proposed a ptrace-based sandbox that protects system calls from being overwritten by copying them to a read-only memory region prior to validating them [64]. A similar solutions could be embedded into GHUMVEE.

7.6 Open Issues

7.6.1 Information Leakage through Covert Channels

As we explained in Section 4.2.3, one possible way to attack GHUMVEE is by means of a covert channel. GHUMVEE can be used as the medium through which this covert channel communicates. For example, the replicae can deliberately delay each other by exploiting the MVEE's lock-step execution mechanism. This mechanism dictates that certain operations may only be completed when all replicae attempt to invoke them. The length of the delay can represent information such as individual bits of a pointer value.

Another possibility is to use the synchronization agents we discussed in Chapter 4 as the medium for the covert channel. A multithreaded master replica could use these agents to leak information to the slave replicae by, e.g., spawning two threads that attempt to enter a critical section protected by the same lock. The master can encode one bit of information by forcing a specific order in which these threads enter the critical section. The synchronization replication agent would replicate this same order in the slave replicae, thus allowing the slaves to receive the encoded information.

Both of these attacks, as well as other possible information leakage attacks could be used to leak information about, e.g., the sending replica's memory layout. Such information could then be used to adapt an attack payload that had previously compromised the sending replica to the receiving replicae's memory layouts. The adapted payload could be sent to the receiving replicae using, e.g., the synchronization replication buffer we discussed in Chapter 4 or the IP-MON replication buffer we discussed in Chapter 6.

Bart Coppens (Ghent University) has written proof-of-concept programs that effectively use our MVEE and both of the mentioned covert channels to pass information from one replica to another. At this point in time, it is unclear whether or not information leakage can be achieved using code-reuse attacks in real-life programs, as DCL prohibits the initial launch of most code-reuse attacks before they can target the mentioned channels. A non-control-data attack that uses a covert channel is a theoretical possibility, however [30].

7.6.2 Internal Information Leakage

Another option to attack GHUMVEE is to compromise the replicae with an attack payload that does not depend on predetermined virtual addresses. Snow et al.'s JIT-ROP is an example of such an attack [124]. JIT-ROP was demonstrated using an attack payload in the form of a javascript code file. This code exploits an information leakage vulnerability in Internet Explorer (IE): in a first phase, the code iteratively maps the entire address space of the IE process to find the locations of useful gadgets. In a second phase, these gadgets are used in a conventional ROP attack.

In the context of GHUMVEE running with DCL, the first phase would collect different addresses in the different replicae. If the collected addresses are used as arguments in system calls, the MVEE will detect this as a divergence. If not, the initial stage of the attack will go by unnoticed, and the attacker might be able to construct malicious payloads in all replicae, with consistent and hence undetected behavior. For this attack to work, the first phase and the construction of the payload for the second phase must invoke identical RVPs in all replicae. It is currently unclear whether this is possible.

Bibliography

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security (CCS'05)*, pages 340–353. ACM, 2005.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13:4:1–4:40, 2009.

[3] Bert Abrath, Bart Coppens, Stijn Volckaert, and Bjorn De Sutter. Obfuscating windows dlls. In *2015 IEEE/ACM 1st International Workshop on Software Protection (SPRO'15)*, pages 24–30, 2015.

[4] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[5] Koenraad MR Audenaert and Luk J Levrouw. Interrupt replay: a debugging method for parallel programs with interrupts. *Microprocessors and Microsystems*, 18(10):601–612, 1994.

[6] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. *Communications of the ACM*, 55(5):111–119, 2012.

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

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

[9] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. CAIN: Silently breaking ASLR in the cloud. In *USENIX Workshop on Offensive Technologies (WOOT)*, WOOT'15, 2015.

[10] Claudio Basile, Zbigniew Kalbarczyk, and Ravi Iyer. A preemptive deterministic scheduling algorithm for multithreaded replicas. In *Proceedings of the 2002 IEEE International Conference on Dependable Systems and Networks (DSN'02)*, pages 149–158, 2002.

[11] Claudio Basile, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Active replication of multithreaded applications. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 17(5):448–465, 2006.

[12] Arkaprava Basu, Jayaram Bobba, and Mark D Hill. Karma: scalable deterministic record-replay. In *Proceedings of the international conference on Supercomputing*, pages 359–368. ACM, 2011.

[13] Pete Becker et al. Working draft, standard for programming language C++. Technical Report N3242=11-0012, Technical Report, 2011.

[14] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, pages 335–348, 2012.

[15] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. *ACM SIGARCH Computer Architecture News*, 38(1):53–64, 2010.

[16] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D Gribble. Deterministic process groups in dOS. In *OSDI*, volume 10, pages 177–192, 2010.

[17] E. Berger, T. Yang, T. Liu, and Gene Novark. Grace: safe multithreaded programming for C/C++. *ACM Sigplan Notices*, 44(10):81–96, 2009.

[18] Emery D Berger and Benjamin G Zorn. Diehard: probabilistic memory safety for unsafe languages. In *ACM SIGPLAN Notices*, number 6, pages 158–168. ACM, 2006.

[19] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. Hacking blind. In *Proc. IEEE Symp. on Security and Privacy (S&P)*, pages 227–242, 2014.

[20] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[21] T.C. Bressoud. TFT: A software system for application-transparent fault tolerance. In *28th Annual Int'l Symp. on Fault-Tolerant Computing*, pages 128–137, 1998.

[22] Thomas C Bressoud and Fred B Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, 1996.

[23] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. Diversified process replicae for defeating memory error exploits. In *IEEE International Performance Computing and Communications Conference*, 2007.

[24] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS'08)*, pages 27–38. ACM, 2008.

[25] Mihai Budiu, Úlfar Erlingsson, and Martín Abadi. Architectural support for software-based protection. In *Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, 2006.

[26] Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.

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

[28] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proc. 17th*

ACM Conf. Computer and Communications Security (CCS), pages 559–572, 2010.

[29] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. Drop: Detecting return-oriented programming malicious code. In *Information Systems Security*, pages 163–177. Springer, 2009.

[30] Shuo Chen, Jun Xu, Emre C Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*. USENIX Association, 2005.

[31] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Proc. Symp. Network and Distributed System Security (NDSS)*, 2014.

[32] Monica Chew and Dawn Song. Mitigating buffer overflows by operating system randomization. Technical report, 2002.

[33] Federal Communications Commission. Measuring broadband America - 2014. `https://www.fcc.gov/reports/measuring-broadband-america-2014`, 2014.

[34] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Symp.*, volume 81, pages 346–355, 1998.

[35] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: a secretless framework for security through diversity. In *Proceedings of the 15th conference on USENIX Security Symposium*, page 9. USENIX Association, 2006.

[36] Stephen Crane, Stijn Volckaert, Felix Schuster, Cristopher Liebchen, Per Larsen, Lucas Davi, Ahmed-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. It's a trap: Table randomization and protection against function-reuse attacks. In *Proceedings of the 22nd ACM conference on Computer and communications security (CCS'15)*. ACM, 2015.

[37] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A Gibson, and Randal E Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*, pages 388–405. ACM, 2013.

[38] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proc. 23rd USENIX Security Symp.*, pages 401–416, 2014.

[39] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proc. of the 6th ACM Symp. on Information, Computer and Communications Security (ASIACCS)*, pages 40–51, 2011.

[40] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: deterministic shared memory multiprocessing. *ACM SIGARCH Computer Architecture News*, 37(1):85–96, 2009.

[41] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. RCDC: a relaxed consistency deterministic computer. *ACM SIGPLAN Notices*, 46(3):67–78, 2011.

[42] EW Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[43] Ulrich Drepper. Selinux memory protection tests, April 2006. http://www.akkadia.org/drepper/selinux-mem.html.

[44] Tyler Durden. Bypassing PaX ASLR protection. *Phrack Magazine*, 59(9):9, 2002.

[45] Isaac Evans, Sam Fingeret, Julián González, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point (er): On the effectiveness of code pointer integrity. In *2015 IEEE Symposium on Security and Privacy (SP)*, 2015.

[46] GNU.org. The gnu c library: Environment access. `http://www.gnu.org/software/libc/manual/html_node/Environment-Access.html`.

[47] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proc. IEEE Symp. on Security and Privacy (S&P)*, pages 575–589, 2014.

[48] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proc. 23rd USENIX Security Symp.*, pages 417–432, 2014.

[49] Ian Goldberg, David Wagner, Randi Thomas, Eric A Brewer, et al. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, volume 6, 1996.

[50] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W Davidson. Ilr: Where'd my gadgets go? In *Proc. IEEE Symp. on Security and Privacy (S&P)*, pages 571–585, 2012.

[51] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. Microgadgets: Size does matter in Turing-complete return-oriented programming. In *Proc. 6th USENIX Workshop on Offensive Technologies (WOOT)*, pages 64–76, 2012.

[52] Petr Hosek and Cristian Cadar. Safe software updates via multi-version execution. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE'13)*, pages 612–621. IEEE Press, 2013.

[53] Petr Hosek and Cristian Cadar. Varan the unbelievable: An efficient n-version execution framework. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*, pages 339–353. ACM, 2015.

[54] Derek R Hower, Polina Dudnik, Mark D Hill, and David A Wood. Calvin: Deterministic or not? free will to choose. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 333–334. IEEE, 2011.

[55] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symposium on Security and Privacy*, S&P'13, pages 191–205, 2013.

[56] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *Usenix Windows NT Symposium*, pages 135–143, 1999.

[57] Galen C. Hunt and James R. Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, 2007.

[58] Intel. Intel 64 and IA-32 architectures software developer's manual volume 2: Instruction set reference, a-z. 2014.

[59] Intel. Intel 64 and IA-32 architectures software developer's manual volume 3B: System programming guide. 2014.

[60] Intel. Intel software guard extensions programming reference. 2014.

[61] Todd Jackson, Andrei Homescu, Stephen Crane, Per Larsen, Stefan Brunthaler, and Michael Franz. Diversifying the software stack using randomized NOP insertion. In *Moving Target Defense II*, pages 151–173. Springer, 2013.

[62] A Jaleel. Memory characterization of workloads using instrumentation-driven simulation–a pin-based memory characterization of the SPEC CPU2000 and SPEC CPU2006 benchmark suites. *VSSAD Technical Report*, 2007. http://www.glue.umd.edu/ ajaleel/workload/.

[63] Ali Jannesari and Walter F Tichy. Identifying ad-hoc synchronization for enhanced race detection. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

[64] Taesoo Kim and Nickolai Zeldovich. Practical and effective sandboxing for non-root users. In *USENIX Annual Technical Conference*, pages 139–144, 2013.

[65] Tim Kornau. Return oriented programming for the ARM architecture. *Master's thesis, Ruhr-Universitat Bochum*, 2010.

[66] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *Proc. 11th USENIX Symp. Operating Systems Design and Implementation (OSDI)*, pages 147–163, 2014.

[67] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[68] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems (TOCS)*, 5(1):1–11, 1987.

[69] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M Chen, and Jason Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. *ACM SIGARCH Computer Architecture News*, 38(1):77–90, 2010.

[70] Luk J Levrouw, Koenraad MR Audenaert, and Jan M Van Campenhout. A new trace and replay system for shared memory programs based on lamport clocks. In *Parallel and Distributed Processing, 1994. Proceedings. Second Euromicro Workshop on*, pages 471–478. IEEE, 1994.

[71] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with ”return-less” kernels. In *Proc. 5th European Conf. Computer Systems*, pages 195–208, 2010.

[72] Linux Programmer's Manual. getauxval(3) - Linux Manual Page.

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

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

[75] Linux Programmer's Manual. signal(7) - Linux Manual Page. `http://man7.org/linux/man-pages/man7/signal.7.html`.

[76] Linux Programmer's Manual. svipc(7) - Linux Manual Page. `http://man7.org/linux/man-pages/man7/svipc.7.html`.

[77] Linux Programmer's Manual. svipc(7) - Linux Manual Page. `http://man7.org/linux/man-pages/man7/svipc.7.html`.

[78] Linux Programmer's Manual. vdso(7) - Linux Manual Page.

[79] Linux Programmer's Manual. wait4(2) - Linux Manual Page. `http://man7.org/linux/man-pages/man2/wait4.2.html`.

[80] Limin Liu, Jin Han, Debin Gao, Jiwu Jing, and Daren Zha. Launching return-oriented programming attacks against randomized relocatable executables. In *2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 37–44. IEEE, 2011.

[81] Tongping Liu, C. Curtsinger, and E. Berger. DTHREADS: efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating System Principles (SOSP'2011)*, pages 327–336, 2011.

[82] Kai Lu, Xu Zhou, Tom Bergan, and Xiaoping Wang. Efficient deterministic multithreading without global barriers. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*, pages 287–300. ACM, 2014.

[83] Jonas Maebe, Michiel Ronsse, and KD Bosschere. Instrumenting jvms at the machine code level. In *3rd PA3CT-symposium*, volume 19, 2003.

[84] The Linux man-pages project. tc-netem(8) - linux manual page. `http://man7.org/linux/man-pages/man8/tc-netem.8.html`.

[85] Matthew Maurer and David Brumley. Tachyon: Tandem execution for efficient live patch testing. In *USENIX Security Symposium*, pages 617–630, 2012.

[86] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *Usenix Security*, page 15, 2006.

[87] William S. McPhee. Operating system integrity in os/vs2. *IBM Systems Journal*, 13(3):230–252, 1974.

[88] John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[89] Timothy Merrifield and Jakob Eriksson. Conversion: Multi-version concurrency control for main memory segments. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*, pages 127–139. ACM, 2013.

[90] Vishwath Mohan, Per Larsen, Stefan Brunthaler, K Hamlen, and Michael Franz. Opaque control-flow integrity. In *Symposium on Network and Distributed System Security (NDSS'15)*, 2015.

[91] John Richard Moser. Virtual machines and memory protections, November 2006. http://lwn.net/Articles/210272/.

[92] Grant Murphy. Position independent executables - adoption recommendations for packages. `https://people.redhat.com/~gmurphy/files/pie.odt`, 2012.

[93] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In *ACM Sigplan Notices*, volume 44, pages 245–258. ACM, 2009.

[94] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *ACM Sigplan Notices*, volume 45, pages 31–40. ACM, 2010.

[95] Nergal. The advanced return-into-lib(c) exploits: PaX case study. `http://phrack.org/issues/58/4.html`.

[96] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. Symp. Network and Distributed System Security (NDSS)*, 2005.

[97] M Olszewski, Jason Ansel, and S Amarasinghe. Kendo: efficient deterministic multithreading in software. *ACM Sigplan Notices*, 44(3):97–108, 2009.

[98] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proc. 26th Annual Computer Security Applications Conf. (ACSAC)*, pages 49–58, 2010.

[99] Pdraig O'Sullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos D. Keromytis. Retrofitting security in COTS software with binary rewriting. volume 354 of

IFIP Advances in Information and Communication Technology, pages 154–172. Springer, 2011.

[100] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proc. IEEE Symp. on Security and Privacy (S&P)*, pages 601–615, 2012.

[101] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Proc. 22nd USENIX Security Symp.*, pages 447–462, 2013.

[102] PaX Team. Address space layout randomization. `http://pax.grsecurity.net/docs/aslr.txt`, 2004.

[103] PaX Team. PaX non-executable pages design & implementation. `http://pax.grsecurity.net/docs/noexec.txt`, 2004.

[104] Niels Provos. Improving host security with system call policies. In *USENIX Security Symposium*, 2002.

[105] Theo De Raadt. tame(2) wip. `http://marc.info/?l=openbsd-tech&m=143725996614627&w=2`.

[106] Ganesan Ramalingam. The undecidability of aliasing. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.

[107] Daan Raman, Bjorn De Sutter, Bart Coppens, Stijn Volckaert, Koen De Bosschere, Pieter Danhieux, and Erik Van Buggenhout. Dns tunneling for network penetration. In *Information Security and Cryptology ICISC 2012*, pages 65–77. Springer Berlin Heidelberg, 2013.

[108] Hans P. Reiser, Jörg Domaschka, Franz J. Hauck, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. Consistent replication of multithreaded distributed objects. In *IEEE Symp. on Reliable Distributed Systems*, pages 257–266, 2006.

[109] Gerardo Richarte et al. Four different tricks to bypass stackshield and stackguard protection. *World Wide Web*, 1, 2002.

[110] Michiel Ronsse and Koen De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Trans. on Computer Systems*, 17(2):133–152, 1999.

[111] J. M. Rushby. Design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 12–21, New York, NY, USA, 1981. ACM.

[112] B Salamat, Andreas Gal, and M Franz. Reverse stack execution in a multi-variant execution environment. In *Workshop on Compiler and Architectural Techniques for Application Reliability and Security*, 2008.

[113] Babak Salamat. *Multi-variant Execution: Run-time Defense Against Malicious Code Injection Attacks*. PhD thesis, University of California at Irvine, Irvine, CA, USA, 2009. AAI3359500.

[114] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European conference on Computer systems (EuroSys'09)*, pages 33–46. ACM, 2009.

[115] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*. IEEE, 2015.

[116] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. Evaluating the effectiveness of current anti-rop defenses. In *Research in Attacks, Intrusions and Defenses*, pages 88–108. Springer, 2014.

[117] Kevin Scott and Jack Davidson. Safe virtual execution using software dynamic translation. In *Computer Security Applications Conf., 2002. Proceedings. 18th Annual*, pages 209–218, 2002.

[118] Kevin Scott, Naveen Kumar, Siva Velusamy, Bruce Childers, Jack W Davidson, and Mary Lou Soffa. Retargetable and reconfigurable software dynamic translation. In *Proc. Int'l Symp. on Code Generation and Optimization (CGO)*, pages 36–47, 2003.

[119] Cedomir Segulja and Tarek S Abdelrahman. What is the cost of weak determinism? In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 99–112. ACM, 2014.

[120] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proc. USENIX Annual Technical Conf. (ATC)*, pages 309–318, 2012.

[121] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security (CCS'07)*, pages 552–561. ACM, 2007.

[122] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proc. 11th ACM Conf. Computer and Communications Security (CCS)*, pages 298–307, 2004.

[123] Jeff Siebert, Hamed Okhravi, and Eric Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *ACM Conference on Computer and Communications Security*, CCS '14, 2014.

[124] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy (SP)*, pages 574–588, 2013.

[125] Solar Designer. Getting around non-executable stack (and fix). `http://seclists.org/bugtraq/1997/Aug/63`, 1997.

[126] László Szekeres, Mathias Payer, Tao Wei, and Dong Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy (SP)*, pages 48–62, 2013.

[127] Francisco J. Torres-Rojas and Mustaque Ahamad. Plausible clocks: constant size logical clocks for distributed systems. *Distributed Computing*, 12(4):179–195, 1999.

[128] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc

attacks. In *Recent Advances in Intrusion Detection*, pages 121–141. Springer, 2011.

[129] Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology, 2005.*, pages 7–12. IEEE, 2005.

[130] Stijn Volckaert, Bart Coppens, and Bjorn De Sutter. Cloning your gadgets: Complete ROP attack immunity with multi-variant execution. *IEEE Trans. on Dependable and Secure Computing*, 2015. To appear, available online at http://ghumvee.elis.ugent.be.

[131] Stijn Volckaert, Bart Coppens, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. Free unlimited calling: Relaxed multi-variant execution. 2015. Manuscript under review.

[132] Stijn Volckaert and Bjorn De Sutter. GHUMVEE website. `http://ghumvee.elis.ugent.be`.

[133] Stijn Volckaert, Bjorn De Sutter, Tim De Baets, and Koen De Bosschere. Ghumvee: efficient, effective, and flexible replication. In *5th International Symposium on Foundations and practice of security (FPS 2012)*, pages 261–277. Springer, 2013.

[134] Stijn Volckaert, Bjorn De Sutter, Koen De Bosschere, and Per Larsen. Multi-variant execution of parallel programs. 2015. Manuscript under review.

[135] John Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies*, 34:43–98, 1956.

[136] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 203–216, 1994.

[137] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 380–395. IEEE, 2010.

[138] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proc. 28th Annual Computer Security Applications Conf. (ACSAC)*, pages 299–308, 2012.

[139] David A Wheeler. Secure programming for linux and unix howto. 1999.

[140] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. RIPE: Runtime intrusion prevention evaluator. In *Pro. 27th Annual Computer Security Applications Conf.*, pages 41–50, 2011.

[141] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, pages 79–93. IEEE, 2009.

[142] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *Software EngineeringESEC/FSE99*, pages 253–267. Springer, 1999.

[143] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P'13)*, pages 559–573. IEEE, 2013.

[144] Mingwei Zhang and R Sekar. Control flow integrity for cots binaries. In *Proceedings of the 22nd USENIX conference on Security*, pages 337–352. USENIX Association, 2013.

[145] Xu Zhou, Kai Lu, Xiaoping Wang, and Xu Li. Exploiting parallelism in deterministic shared memory multiprocessing. *Journal of Parallel and Distributed Computing*, 72(5):716–727, 2012.