Cloning your Gadgets: Complete ROP Attack Immunity with Multi-Variant Execution

Stijn Volckaert, Bart Coppens, and Bjorn De Sutter, *Member, IEEE Computer Society*

***Abstract*—In this paper, we present Disjoint Code Layouts (DCL), a technique that complements Multi-Variant Execution [1] and W⊕X protection to effectively immunize programs against control flow hijacking exploits such as Return Oriented Programming (ROP) [2] and return-to-libc attacks [3]. DCL improves upon Address Space Partitioning (ASP), an earlier technique presented to defeat memory exploits. Unlike ASP, our solution keeps the full virtual address space available to the protected program. Additionally, our combination of DCL with Multi-Variant Execution is transparent to both the user and the programmer and incurs much less overhead than other ROP defense tools, both in terms of run time and memory footprint.**

***Index Terms*—return oriented programming, return-to-libc, replication, monitoring, memory exploits, overhead, protection**

I. Introduction

Hackers keep finding new vulnerabilities in major software packages at an astonishing rate. Coincidentally, exploit prevention has been an active research topic for over a decade, with many countermeasures being proposed.

Of the many proposed techniques, only a handful have been successfully deployed in mainstream operating systems and compilers, i.e., the techniques that incur limited run-time overhead and that require little to no cooperation from application developers and users. These include Address Space Layout Randomization [4] and W⊕X [5]. Additionally, nearly every modern compiler enables stack overflow protection [6] by default. Despite their success in terms of deployment rate, these techniques only raise the bar for hackers. Over the years, each of them has been bypassed or hacked.

In 2007 Shacham presented the first Return Oriented Programming (ROP) attacks for the x86 architecture [7]. 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 [8], ARM [9], 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.

All authors are with the Computer Systems Lab, Department of Electronics and Information Systems, Ghent University, Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium. E-mail: {stijn.volckaert, bart.coppens, bjorn.desutter}@elis.ugent.be.

The authors thank the Agency for Innovation by Science and Technology in Flanders (IWT) and the Fund for Scientific Research - Flanders.

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 before any divergent I/O operation is requested, the monitor can effectively block any ROP attack before it can cause harm. Our design and prototype 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 existing 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. Not requiring special compiler support, our solution is compatible with existing compilers, including their support for, e.g., stack protection.
- Requiring no or limited, almost 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 code and data address-dependent behavior.
- Unlike some existing techniques, DCL causes only marginal memory footprint overhead within the protected application's address space. As such, DCL can protect programs that flirt with address space boundaries on systems with small address spaces, such as 32-bit Linux systems. System-wide, DCL does cause considerable memory overhead due to its duplication of process-local data regions such as the heap, and of private writable mmaped pages such as mutable data

sections. 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.

The remainder of this paper is organized as follows. Section II discusses existing attacks and defenses for a range of exploits. Section III then presents our solution's top-level design. Section IV discusses how to replicate real-world programs with memory layout diversification, and Section V presents our approach to generate completely disjoint code layouts in replicated programs. Our solution's overhead is evaluated in Section VI. Section VII concludes.

II. 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++ [10]. 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.

A. Memory Attacks and Defenses

Every modern operating system supports at least Address Space Layout Randomization [4] and W⊕X [5]. Additionally, nearly every modern compiler enables stack overflow protection [6] 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 [11] and brute-force attacks [12]. 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 [13]. Additionally, Bittau et al. recently demonstrated that even on 64-bit platforms, ASLR brute-force attacks are feasible [14].

W⊕X has not been attacked directly. It can however be bypassed easily. Solar Designer demonstrated return-to-libc attacks as early as 1997 [3], long before W⊕X and its predecessor, non-executable stacks [15], 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 [16].

In 2007, Shacham presented the first Return Oriented Programming (ROP) attacks [7]. 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 [17]. 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 [7].

ROP attacks were later generalized to other architectures such as SPARC [8], ARM [9], 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.

B. Custom Code Analysis and Code Generation

Dynamic instrumentation tools such as DROP [18] and ROPdefender [19] 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 [20]. TaintCheck does however suffer from large run-time overhead up to 2500%.

Other tools based on dynamic binary translation rewrite a program completely. Hiser et al. [21] proposed Instruction Location Randomization (ILR), a technique implemented in the Strata VM [22]. 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 [23], [24], [25] 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 [26].

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 [27]. 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 at all [17], [28].

Onarlioglu et al. presented a similar but more promising technique: G-Free [29]. 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. [30] 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 [6]. LibSafe and many standard C-libraries offer protection against format string vulnerabilities through hardened versions of string functions [31]. Control-flow integrity (CFI) techniques add checks around indirect jumps to detect unintended branch targets [32], [33]. As shown by Göktaş et al. [34], Davi et al. [35] and several others, even the strictest, most fine-grained of CFI enforcement policies in use do not mitigate ROP attacks completely.

The most recent contribution to this domain is Code-Pointer Integrity (CPI) [36]. 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 [37]. 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 Address-Sanitizer (ASan) [38]. 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 [39]. 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 [40] and REINS [41], are also explicitly limited to non-obfuscated code.

C. 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 [1] and the conceptually similar Multi-Variant Execution Environments [42], [43], [44], [45]. 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 [42], [46]. These replicae are generated with a modified version of GCC, with the replicae with stacks growing upwards being only marginally slower. This technique mitigates even the most advanced stack smashing attacks that bypass other stack protectors [47].

Salamat et al. also proposed to renumber system calls [48]. 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; see Section IV.

Cox et al. [1] and Cavallaro [45] 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.

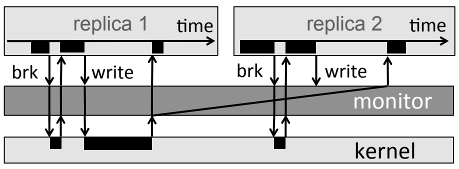


Fig. 1: Basic operation of a MVEE

Cox et al. also proposed instruction set tagging as a defense mechanism against code injection [1]. In an offline step, a binary rewriter [49] 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 [50]. 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.

In 2012, we presented GHUMVEE, a N-variant monitor that supports a wide range of diversification between replicae, including code diversification and ASLR [43]. These forms of diversification are supported even for multi-threaded applications that feature address-dependent behavior and non-deterministic thread synchronization [44]. The techniques presented in this paper build on GHUMVEE, so we will discuss its internal operation in more detail in the next section. As we previously reported, N-Variant Systems can achieve a limited average execution time overhead of 16% [43]. As the evaluation in Section VI-C will show, our current, more optimized version of GHUMVEE, can replicate programs at very limited execution time overheads of only 6.37% on the AMD64 platform.

III. Multi-Variant Execution with Disjoint Code Layouts

As we've mentioned in the previous section, the technique presented in this paper builds on GHUMVEE, our prototype tool for Multi-Variant Execution [43], [44].

Rather than launching an application directly, a user seeking protection against ROP and other memory-based attacks will invoke the GHUMVEE monitor to replicate the application. Our implementation supports the i386 and AMD64 architectures for the GNU/Linux platform but there are no fundamental restrictions to either the architectures, or the platform: All design options we lifted for GHUMVEE target applications running on top of an unmodified OS running on a commercial off-the-shelf multi-core processor.

Upon invocation, GHUMVEE's monitor launches two or more replicae of the application, attaches itself to those replicae like a debugger, and feeds them their original arguments. The monitor is responsible for ensuring that the replication is transparent, i.e., that with the exception of timing, the replication does not influence the I/O behavior of the application as observed by the user. To that end, the monitor intercepts all I/O between the replicae and their environment. Fig. 1 illustrates this for two system calls, brk and write. For a call like brk, the monitor waits until both replica have issued the call before passing them through to

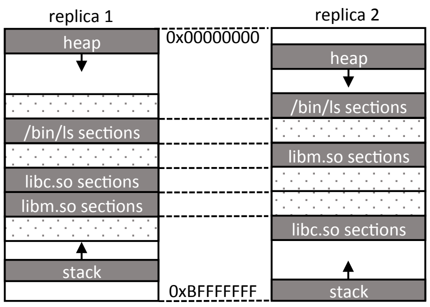


Fig. 2: Two replicae's address spaces with disjoint code layouts.

	class 1 master call	class 2 sanity	class 3 diversity	class 4 synchronization
functionality				
type	system call	system call	procedure	procedure & macro
I/O related	yes	no	no	no
side-effect	no	yes	no	no
examples	read, write, getpid, gettimeofday	brk, mprotect	pointer-sensitive hash	mutex_lock
disposition				
replicated by	monitor	-	master replica	master replica
executed by	master replica	all replicae	master replica	all replicae
transparency				
technique	ptrace	ptrace	interposer	code patching
user transparency	yes	yes	yes	yes
developer transparency	yes	yes	mostly	partially

TABLE I: Classification of rendez-vous points

the kernel to allocate memory for both replicae. After the kernel has processed the system calls, the monitor passes the resulting pointers back to the replicae. Of the two write calls by the two replicae, the monitor passes only one to the kernel, as the external world should observe only one write operation. The result of the passed call is then replicated by the monitor and passed to both replicae, at which point they continue executing. During the replicae's execution, the monitor checks their I/O to ensure that they behave identically. Whenever an I/O divergence is detected among the replicae, the monitor signals this and halts execution before the I/O operation is passed on to the environment. For that reason, the replicae are synchronized at all I/O operations they initiate, as shown in Fig. 1.

The core idea behind DCL is to diversify the code layouts of the two or more replicae, such that ROP attacks become asymmetrical. All ROP attacks we are aware of have in common that they rely on addresses of executable code, i.e., so-called gadgets, to hijack the control over the program under attack, i.e., to steer the flow of control between gadgets in the program. These addresses are hard-coded or computed when launching the attack, and passed to the program as part of its input. Because MVEEs like GHUMVEE replicate the inputs to all replicae, all of them get the same input, including the same set of gadget addresses. So in order to protect a replicated application from ROP attacks, i.e., to ensure that a ROP attack leads to diverging behavior of the replicae, it suffices to ensure that no gadgets occur at the same addresses in multiple replica.

In theory, it can even suffice that no equivalent gadgets occur at the same addresses in multiple replica. Since equivalence is hard to prove or disprove, however, in particular without requiring time-consuming code analyses, we opted for the practical approach: GHUMVEE ensures that no gadgets occur at the same addresses in multiple replica, simply by enforcing that at each virtual address in the replicae's address-spaces, at most one replica actually maps code. Figure 2 shows possible address space layouts with disjoint code (and statically allocated data) layouts for two replica of a simple example command-line tool like ls.

As we will discuss in Section IV, GHUMVEE also supports other forms of diversification such as ASLR. These can be combined with DCL to also enforce diversified stack and heap layouts.

IV. Diversifying Multi-Variant Execution

To replicate applications, GHUMVEE spawns 2 or more replica processes to which it attaches itself using Linux' ptrace API [51]. From then on, GHUMVEE acts as a proxy between the replicae and the kernel as depicted in Figure 1. In this section, we present GHUMVEE's overall design and concepts, focusing on those aspects that are necessary to support diversified replicae and that enable GHUMVEE to enforce disjoint code layouts.

A. Rendez-vous Points

GHUMVEE uses a master/slave model for replication. Replicae run independently in parallel, but when they reach so-called rendez-vous points (RVPs), the monitor suspends them. The monitor can then interfere by inspecting and manipulating their state. GHUMVEE only allows replicae to pass a RVP if they are in consistent states. When all replicae are suspended, e.g., at the entrance of a system call, their arguments must be equivalent. If not, GHUMVEE raises an alarm. Cox and Salamat provide a formal definition of system call argument equivalence [1], [42].

GHUMVEE differentiates four types of RVPs based on their properties and purpose. Table I summarizes the classification of RVPs based on their properties. RVP classes 1 and 2 serve to compare the replicae's behavior as well as to ensure consistent replication. Classes 3 and 4 only serve to ensure consistent replication, for which GHUMVEE must feed all replicae the same input and it must impose synchronization determinism.

1) Master Calls: First, entrances and exits of I/O-related system calls are intercepted. In this case, "I/O-related" has to be interpreted broadly. For example, asking the OS for one's process ID is to be considered I/O. The execution of the replicae at those points is intercepted with the ptrace API, which hence requires no intervention or code patching by the application programmer. At these RVPs, GHUMVEE ensures that only the master replica performs the I/O, hence the name master call. The result of the operation is then replicated to the slave replicae. Since all replicae are I/O-equivalent, proper handling of I/O-related RVPs suffices to replicate simple single-threaded programs. In the context of this paper, I/O-equivalent means that arguments at system call invocations should be identical, unless they are pointer values or indices pointing to buffers or structures, in which case their contents should be (similarly) equivalent. For comparing and replicating system call arguments, the ptrace API is extremely slow. The latest versions of GHUMVEE therefore rely on the process_vm_* API, which includes two system calls that enable direct inter-process copying of arbitrarily sized memory blocks since Linux 3.2.

2) Sanity Checks: Some system calls are not replicated by the monitor because they have side effects that are required in all replicae. So all replicae execute them, and the monitor intercepts them for intrusion detection, for sanity checking, and for enforcing consistent memory allocation behavior. This interception is also handled fully transparently with the ptrace API. Examples are the brk and mprotect system calls to allocate and protect memory.

3) Diversity Replication: GHUMVEE does not require replicae to be fully identical, it only requires I/O-equivalence. In some cases, this relaxed requirement leads to behavioral differences between the replicae, not with respect to their I/O behavior, but with respect to the other RVPs that get executed. We have observed many programs that feature such behavioral differences based on the heap lay-out. One recurring example is the use of heap pointer values in the computation of hash keys for hash tables or search trees. For example, the run-time decision to resize hash tables is often based on the number of observed hash collisions, which depends on the actual hash values. When the hash values diverge in the replicae, the timing of resizing operations will diverge, as will the involved allocation of memory, including system calls and synchronization. Either the monitor needs to be extended to support such differences in behavior, or the differences have to be eliminated. Because the former would make the monitor's intrusion detection much more complex, and thus result in unacceptable overhead, we opted for the second solution. Using interposers, the agent intercepts the execution of sensitive procedures in the replicae, and replicates the behavior of the master in the slaves [52], [53]. When, e.g., a hash key is computed in the master replica based on the pointer values in that replica, the procedure computing the hash in the master is interposed to extract the computed hash. The interposer then passes that value to the slaves' interposers, where it replaces the hash value computed in those slaves.

To support these RVPs, the programmer must ensure that all address-dependent behavior is interposable and that computed values can be identified in the replicae. This means that the computations of, e.g., hashes should be bound to identifiable functions that return arithmetic values computed on the pointer values. In C code, for example, those functions should not be declared static, but be declared noinline instead. Furthermore, a list of those functions should be provided to the GHUMVEE monitor. Apart from that, no developer effort is required. We refer to previous work for a more extensive discussion of the replication of address-sensitive behavior and an assessment of the required programmer effort [43], [44].

4) Synchronization Replication: GHUMVEE can also intervene on synchronization RVPs. On commodity OSs, thread interleavings are non-deterministic by nature. Enabling replication of multi-threaded programs thus requires the replication agent to either enforce the same thread interleavings among all replicae, or, in the absence of data races, to enforce the same order in which related critical sections are entered, locks are taken, and atomic operations are executed. GHUMVEE can therefore intervene in the execution of all synchronization operations. This includes high-level operations such as pthread mutexes, as well as low-level operations such as CAS-based spinlocks and atomic operations. To this end, we added RVPs to all atomic operations in eglibc (http://www.eglibc.org). At these RVPs, the monitor forces the master replica to record the order of all the synchronization operations it performs. At the corresponding RVPs in the slaves' threads, the monitor forces those threads to check the recorded information and to stall if necessary, i.e., to stall until the necessary information is recorded by the master to enforce the same ordering and synchronizations decisions in the slave.

The GHUMVEE-enabled version of eglibc does not need to replace the standard libc installed on a system. It is be shipped with GHUMVEE itself and is injected transparently into a replicated application by manipulating the arguments of sys_execve calls. Section V-B describes a similar technique for enforcing disjoint code layouts. For a detailed overview of the implementation of our GHUMVEE-enabled eglibc, we refer to our other work [44].

B. Support for Other Forms of Non-Determinism

Many programs exhibit non-deterministic behavior, e.g., because of the synchronization-related non-determinism mentioned in Section IV-A4.

There are, however, many other sources of non-determinism, even in trivial programs. Most of these sources can be elegantly eliminated by GHUMVEE. Salamat gave an extensive overview of several sources including asynchronous signals, file descriptors, process IDs and random number generators [42]. On top of Salamat's solutions, we already proposed solutions for time stamp counters, shared memory and reading from the /proc interface [43].

One last source of non-determinism that has not been described in earlier work is the VDSO. Both the x86_64 version and recent i386 versions (> 3.15) of the Linux kernel export a VDSO that contains a memory page with timing information [54]. This memory page is shared among all running processes and is periodically updated by the kernel to provide every running program with a reliable source of timing information that can be accessed at a very low cost. At the time of writing, the x86_64 version of (e)glibc implements gettimeofday, clock_gettime and time functions that access this timing page, thereby eliminating the need to perform costly system calls. Unfortunately, if a replica omits the system call invocations in these functions, GHUMVEE cannot provide consistent input. We have therefore chosen to hide the VDSO on platforms that contain the timing page. We hide the VDSO by deleting the AT_SYSINFO_EHDR entry from the ELF auxiliary headers when the replicae start up. Deleting this entry is a trivial extension of the loader program described in Section V-C.

C. Limitations of Multi-Variant Execution

Even though there are many issues that arise when implementing an MVEE, there are very few fundamental limitations to the programs that can be run inside an MVEE. The only fundamental problem that we see involves bi-directional shared memory. On the Linux platform, the kernel exposes two interfaces that may be used to map memory pages that are shared with other processes. When such pages are used though, programs can communicate with each other directly, without using system calls. This direct communication cannot be reconciled with replicating I/O at the system call level and GHUMVEE therefore does not allow its replicae to set up shared pages. Instead, GHUMVEE returns the EPERM error value to the replicae as if the kernel refused to perform the requested mapping. As we described in previous work, this is not a major problem for most applications: Many applications requesting such shared memory can handle the fact that even on a native system, the kernel might refuse such mappings. These applications handle it by offering less efficient, but working fall-back alternatives [43]. While a more elegant solution would be desirable for GHUMVEE, it should be clear that our current solution does not undermine the provided security guarantees in any way: any application requesting bi-directional shared memory under the control of GHUMVEE will simply not get it.

V. 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 V-A 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.

A. 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 3.

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. Alternatively, one can 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 executables 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 [55].

If the executable is dynamically linked, the kernel then maps an architecture-specific virtual dynamic shared object

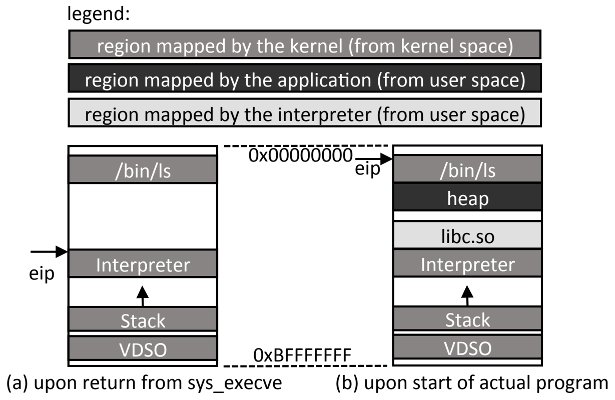

legend:
region mapped by the kernel (from kernel space)
region mapped by the application (from user space)
region mapped by the interpreter (from user space)
/bin/ls
Interpreter
eip
Stack
VDSO
0x00000000
0xBFFFFFFF
eip
/bin/ls
heap
libc.so
Interpreter
Stack
VDSO
(a) upon return from sys_execve
(b) upon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

Fig. 3: Address space layout for standard invocation of the ls tool.

(VDSO) into memory. The VDSO may contain specialized code to transfer control from user space to kernel space or it may contain 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 by ASLR.

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

Figure 3(a) depicts the process address space layout after return from the sys_execve call. For the sake of completeness, Figure 3(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.

B. 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.

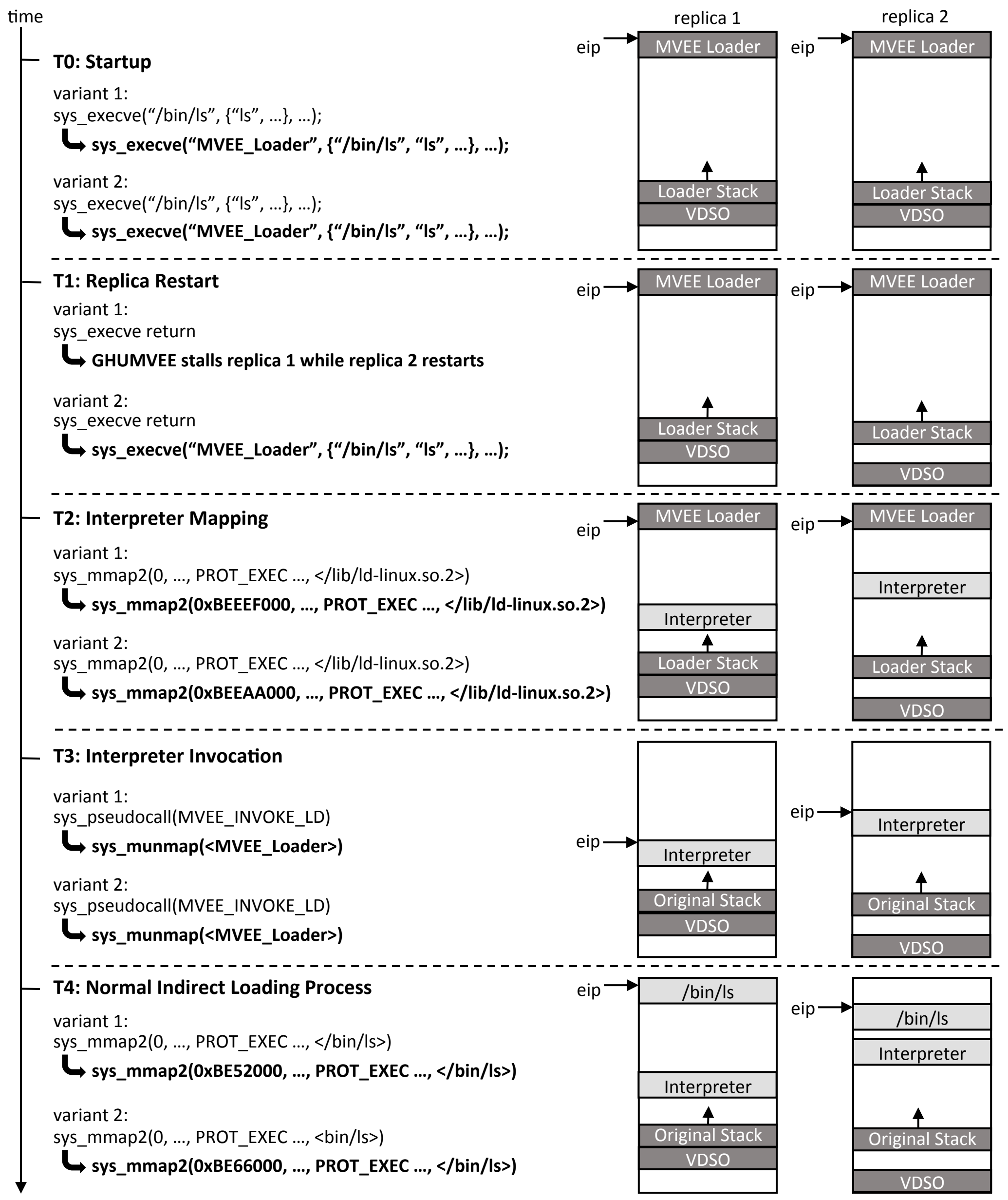
C. Program Interpreter

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 V-A, 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 4. 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 4 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,



```
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>)
```

Fig. 4: Address space snapshots during GHUMVEE's DCL program launching.

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 4 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 [56]. The result is depicted on the right in Figure 4 at time **T2**.

The MVEE Loader then transfers control to GHUMVEE through a pseudo-system call, as depicted on the left of Figure 4 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. The replicae then have the memory layout as depicted on the right of Figure 4 at time **T3: Interpreter Invocation**.

The interpreter will then continue to load and map the original program and the dynamically linked libraries, of which all mapping will again be intercepted and manipulated to enforce DCL, as shown on the left of Figure 4 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 4 at time **T4**.

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.

D. Disjoint Code Layout vs. Address Space Partitioning

As mentioned in Section II, Cox et al. and Cavallaro independently proposed to combat memory exploits with essentially identical techniques they called Address Space Partitioning (ASP) [1] and Non-Overlapping Address Spaces [45] 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 V-A.

Finally, whereas DCL relies on Position Independent Executables (PIE) [55] 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.

E. 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 [57], [58]. 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 Section IV-C, 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⊕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.

F. 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) [59], 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.

VI. 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.

A. 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.

B. 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 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.

C. 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 included the benchmark results for 3 and 4 replicae for completeness' sake.

As shown in Figures 5 and 6, 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 [60].

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 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 discussed in previous work [44], 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.

[1]For using hardware support, MPlayer tries to obtain shared memory pages with read and write permissions from the kernel. As explained in Section IV-B, GHUMVEE can currently not support the potential bi-directional communication through shared memory with such permissions. As explained in our previous work [43], GHUMVEE therefore intercepts the system call and returns an error value to indicate that the allocation requested to the kernel failed [43]. MPlayer then falls back on its software-only version.

[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.

D. 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.

E. 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. [14]. 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 libary. 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.

VII. Conclusions

In this paper, 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.

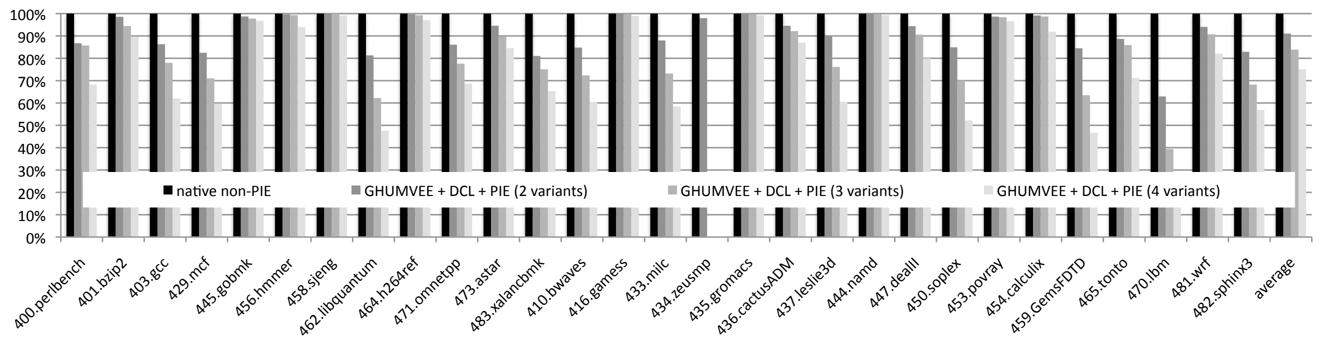


Fig. 5: Relative performance of 32-bit protected SPEC 2006 benchmarks.

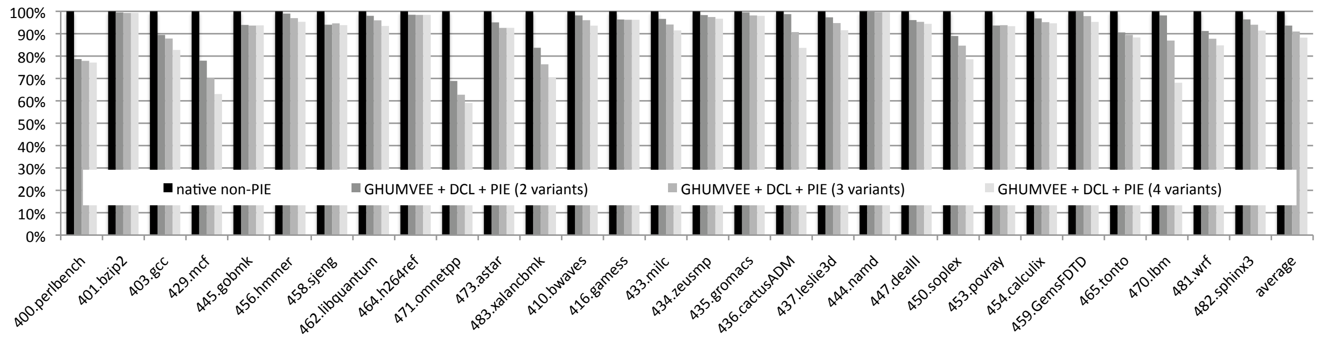


Fig. 6: Relative performance of 64-bit protected SPEC 2006 benchmarks.

References

[1] B. Cox, D. Evans, *et al.*, "N-variant systems: A secretless framework for security through diversity," in *Proc. 15th USENIX Security Symp.*, 2006, pp. 105–120.

[2] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. on Information and System Security (TISSEC)*, vol. 15, no. 1, p. 2, 2012.

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

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

[5] ——, "PaX non-executable pages design & implementation," http://pax.grsecurity.net/docs/noexec.txt, 2004.

[6] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, *et al.*, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. 7th USENIX Security Symp.*, vol. 81, 1998, pp. 346–355.

[7] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. 14th ACM Conf. Computer and Communications Security (CCS)*, 2007, pp. 552–561.

[8] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: generalizing return-oriented programming to RISC," in *Proc. 15th ACM Conf. Computer and Communications Security (CCS)*, 2008, pp. 27–38.

[9] T. Kornau, "Return oriented programming for the ARM architecture," *Master's thesis, Ruhr-Universitat Bochum*, 2010.

[10] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proc. IEEE Symp. on Security and Privacy (S&P)*, 2013, pp. 48–62.

[11] T. Durden, "Bypassing PaX ASLR protection," *Phrack Magazine*, vol. 59, no. 9, p. 9, 2002.

[12] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proc. 11th ACM Conf. Computer and Communications Security (CCS)*, 2004, pp. 298–307.

[13] L. Liu, J. Han, D. Gao, J. Jing, and D. Zha, "Launching return-oriented programming attacks against randomized relocatable executables," in *Proc. 10th IEEE Int'l Conf. Trust, Security and Privacy in Computing and Communications*, 2011, pp. 37–44.

[14] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, "Hacking blind," in *Proc. IEEE Symp. on Security and Privacy (S&P)*, 2014, pp. 227–242.

[15] Solar Designer, "Non-executable stack patch," http://openwall.com/linux/, 1998.

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

[17] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz, "Microgadgets: Size does matter in Turing-complete return-oriented programming," in *Proc. 6th USENIX Workshop on Offensive Technologies (WOOT)*, 2012, pp. 64–76.

[18] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "Drop: Detecting return-oriented programming malicious code," in *Information Systems Security*. Springer, 2009, pp. 163–177.

[19] L. Davi, A.-R. Sadeghi, and M. 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)*, 2011, pp. 40–51.

[20] J. Newsome and D. X. 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.

[21] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "Ilr: Where'd my gadgets go?" in *Proc. IEEE Symp. on Security and Privacy (S&P)*, 2012, pp. 571–585.

[22] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa, "Retargetable and reconfigurable software dynamic translation," in *Proc. Int'l Symp. on Code Generation and Optimization (CGO)*, 2003, pp. 36–47.

[23] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing." in *Proc. 22nd USENIX Security Symp.*, 2013, pp. 447–462.

[24] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attacks," in *Proc. Symp. Network and Distributed System Security (NDSS)*, 2014.

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

[26] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to

prevent code-reuse attacks is hard," in *Proc. 23rd USENIX Security Symp.*, 2014, pp. 417–432.

[27] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with "return-less" kernels," in *Proc. 5th European Conf. Computer Systems*, 2010, pp. 195–208.

[28] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proc. 17th ACM Conf. Computer and Communications Security (CCS)*, 2010, pp. 559–572.

[29] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free: defeating return-oriented programming through gadget-less binaries," in *Proc. 26th Annual Computer Security Applications Conf. (ACSAC)*, 2010, pp. 49–58.

[30] T. Jackson, A. Homescu, S. Crane, P. Larsen, S. Brunthaler, and M. Franz, "Diversifying the software stack using randomized NOP insertion," in *Moving Target Defense II*. Springer, 2013, pp. 151–173.

[31] A. Baratloo, N. Singh, and T. K. Tsai, "Transparent run-time defense against stack-smashing attacks." in *USENIX Annual Technical Conf., General Track*, 2000, pp. 251–262.

[32] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proc. 12th ACM Conf. Computer and Communications Security (CCS)*, 2005, pp. 340–353.

[33] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proc. 22nd USENIX Security Symp.*, 2013, pp. 337–352.

[34] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proc. IEEE Symp. on Security and Privacy (S&P)*, 2014, pp. 575–589.

[35] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *Proc. 23rd USENIX Security Symp.*, 2014, pp. 401–416.

[36] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proc. 11th USENIX Symp. Operating Systems Design and Implementation (OSDI)*, 2014, pp. 147–163.

[37] G. Ramalingam, "The undecidability of aliasing," *ACM Trans. on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1467–1471, 1994.

[38] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. USENIX Annual Technical Conf. (ATC)*, 2012, pp. 309–318.

[39] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proc. IEEE Symp. on Security and Privacy (S&P)*, 2012, pp. 601–615.

[40] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis, "Retrofitting security in COTS software with binary rewriting," ser. IFIP Advances in Information and Communication Technology, vol. 354. Springer, 2011, pp. 154–172.

[41] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Securing untrusted code via compiler-agnostic binary rewriting," in *Proc. 28th Annual Computer Security Applications Conf. (ACSAC)*, 2012, pp. 299–308.

[42] B. Salamat, "Multi-variant execution: Run-time defense against malicious code injection attacks," Ph.D. dissertation, University of California at Irvine, 2009.

[43] S. Volckaert, B. De Sutter, T. De Baets, and K. De Bosschere, "GHUMVEE: Efficient, Effective, and Flexible Replication," in *Proc. 5th Int'l Symp. on Foundations & Practice of Security*, 2012, pp. 261–277.

[44] S. Volckaert, B. De Sutter, and K. De Bosschere, "Replicatable determinism for parallel programs," 2015, Manuscript in preparation, http://users.elis.ugent.be/ svolckae/determinism/.

[45] L. Cavallaro, "Comprehensive memory error protection via diversity and taint-tracking," Ph.D. dissertation, Univ. Degli Studi Di Milano, 2007.

[46] B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space," in *Proc. 4th ACM European Conf. Computer Systems (EuroSys)*, 2009, pp. 33–46.

[47] Aleph One, "Smashing the stack for fun and profit," *Phrack Magazine*, vol. 7, no. 49, 1996.

[48] B. Salamat, A. 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.

[49] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere, "Diablo: a reliable, retargetable and extensible link-time rewriting framework," in *Proc. 5th IEEE Int'l Symp. on Signal Processing and Information Technology (ISSPIT)*, 2005, pp. 7–12.

[50] K. Scott and J. Davidson, "Safe virtual execution using software dynamic translation," in *Computer Security Applications Conf., 2002. Proceedings. 18th Annual*, 2002, pp. 209–218.

[51] Linux Programmer's Manual, "ptrace(2) - Linux Manual Page."

[52] G. Hunt and D. Brubacher, "Detours: Binary Interception of Win32 Functions," in *Proc. 3rd USENIX Windows NT Symp.*, 1999, p. 14.

[53] T. W. Curry, "Profiling and tracing dynamic library usage via interposition," in *Proc. USENIX Summer 1994 Tech. Conf.*, 1994, pp. 267–278.

[54] Linux Programmer's Manual, "vdso(7) - Linux Manual Page."

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

[56] Linux Programmer's Manual, "getauxval(3) - Linux Manual Page."

[57] J. R. Moser, "Virtual machines and memory protections," November 2006, http://lwn.net/Articles/210272/.

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

[59] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: Runtime intrusion prevention evaluator," in *Pro. 27th Annual Computer Security Applications Conf.*, 2011, pp. 41–50.

[60] 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/.

Stijn Volckaert is a Ph.D. student at Ghent University in the Computer Systems Lab. He obtained his BEng. degree in Computer Science from Ghent University's Faculty of Engineering in 2008 and his Msc. degree in Computer Science from Ghent University's Faculty of Engineering in 2010. His research focuses on software anti-tampering.

Bart Coppens is a postdoctoral researcher at Ghent University in the Computer Systems Lab. He obtained his Ph.D. degree in Computer Science from Ghent University's Faculty of Engineering in 2013. His research focuses on the use of compiler techniques for software protection, including obfuscation and diversification.

Bjorn De Sutter is a professor at Ghent University in the Computer Systems Lab. He obtained his Msc. and Ph.D. degrees in Computer Science from Ghent University's Faculty of Engineering in 1997 and 2002. His research focuses on the use of compiler techniques to aid programmers with non-functional aspects of their software, such as performance, code size, reliability, and software protection.