# The Continuing Arms Race

# 8 Multi-Variant Execution Environments

**Bart Coppens, Bjorn De Sutter, Stijn Volckaert**

Memory corruption vulnerabilities are a common problem in software implemented in C/C++. Attackers can exploit these vulnerabilities to steal sensitive data and to seize or disrupt the system on which the software is executed. Memory safety techniques can, in principle, eliminate these vulnerabilities [Nagarakatte et al. 2009, Nagarakatte et al. 2010] but are prohibitively expensive in terms of runtime overhead [Szekeres et al. 2013].

Instead, modern operating systems and compilers deploy exploit mitigations such as Address Space Layout Randomization (ASLR) [PaX Team 2004a], Data Execution Prevention (DEP, a.k.a. W⊕X) [PaX Team 2004b], and stack canaries [Cowan et al. 1998]. These exploit mitigations incur minimal performance overhead, but are limited in scope—often only defending against one particular type of exploit—and can be bypassed with only modest effort.

Up-and-coming exploit mitigations, such as control-flow integrity [Abadi et al. 2005a, Tice et al. 2014], require more effort to bypass [Göktas et al. 2014a, Davi et al. 2014, Carlini et al. 2015e, Evans et al. 2015, Schuster et al. 2015], but, similar to the aforementioned defenses, they defend only against attacks of one particular type: code reuse.

The ubiquity of multi-core processors has made Multi-Variant Execution Environments (MVEEs) an increasingly attractive option to provide strong, comprehensive protection against memory corruption exploits, while still incurring only a fraction of the runtime overhead of full memory safety. MVEEs have been shown to successfully defend against several types of attacks, including code reuse [Volckaert et al. 2015], information leakage [Koning et al. 2016], stack buffer overflows [Salamat et al. 2009], and code injection [Cox et al. 2006].

The underlying idea is to run several diversified instances of the same program, often referred to as variants or replicas, side by side on equivalent program inputs. The MVEE's main component, the monitor, feeds all variants these equivalent inputs and monitors the variants' behavior. The diversity techniques used to generate the variants ensure that the variants respond differently to malicious inputs, while leaving the behavior under normal operating conditions unaffected. The MVEE monitor detects the diverging behavior and halts the execution of the variants before they can harm the system. This implies that the variants must, to some extent, be executed in lockstep: potentially harmful operations in a variant are only executed when the consistency with the other variants has been validated.

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. In this chapter, we discuss the design of MVEEs and provide implementation details about our own MVEE, the Ghent University Multi-Variant Execution Environment, or GHUMVEE, and its extensions. GHUMVEE has been open sourced and can be downloaded from http://github.com/stijn-volckaert/ReMon/.

## 8.1 General Design of an MVEE

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

### 8.1.1 Monitoring Granularity

Monitoring the variants' 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 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. On modern operating systems that offer page-level memory protection, each 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 variants' behavior. The premise of multi-variant execution is that the variants are constructed such that they react differently to malicious input. While a given malicious input might be sufficient to seize control of one specific variant, it will not have the desired effect on other variants. These other variants 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 [Salamat et al. 2009, Cox et al. 2006].

### 8.1.2 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 variants and the monitor. Since most monitors intervene in each system call invocation, such interactions can occur frequently.

All existing monitors interact synchronously with the variants. When a variant instigates an interaction with the monitor, it must wait until the monitor returns the control flow to the variant 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-variant 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 variant to the monitor. Context switches are notoriously slow as they require a page table and a Translation Lookaside Buffer (TLB) flush [Belay et al. 2012]. CP monitors can therefore be detrimental for the variants' performance.

The advantage of CP monitors is that they are portable and easy to implement because they rely solely on the operating system's standardized debugging interfaces, and they are strongly isolated from the variants, since address spaces form a hardware-enforced boundary between processes. Placing the monitor outside the variants' address spaces therefore protects it from misbehaving variants. Table 8.1 illustrates that most authors recognize the importance of such a hardware-enforced boundary. Almost all the existing monitors prioritize security over performance

**Table 8.1** **Classification of Existing MVEEs Based on Their Position in the Software Stack**

| | Unprivileged Level (UL) | Privileged Level (PL) |
|---|---|---|
| In-Process (IP) | VARAN [Hosek and Cadar 2015] | N-Variant Systems [Cox et al. 2006]<br>RAVEN [Co et al. 2016]<br>MvArmor [Koning et al. 2016] |
| Cross-Process (CP) | DieHard [Berger and Zorn 2006]<br>Cavallaro [Cavallaro 2007]<br>Orchestra [Salamat et al. 2009]<br>Tachyon [Maurer and Brumley 2012]<br>Mx [Hosek and Cadar 2013]<br>GHUMVEE [Volckaert et al. 2013] | |
| IP+CP | | ReMon [Volckaert et al. 2016] |

and run cross-process. These monitors correspond with the label "CP/UL" (Cross-Process/Unprivileged Level) in Figure 8.1.

N-Variant Systems [Cox et al. 2006], RAVEN [Co et al. 2016], and MvArmor [Koning et al. 2016] are notable exceptions. These monitors run within the same address space as the variants (In-Process, or IP) but are protected from misbehaving variants because the monitors operate at a higher privilege level (kernel level for N-Variant Systems and RAVEN, supervisor level for MvArmor). This design is represented by "IP/PL" (In-Process/Privileged Level) in Figure 8.1. 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 [Rushby 1981]. An additional disadvantage is that the programming interfaces that are available at the privileged level are non-standardized and

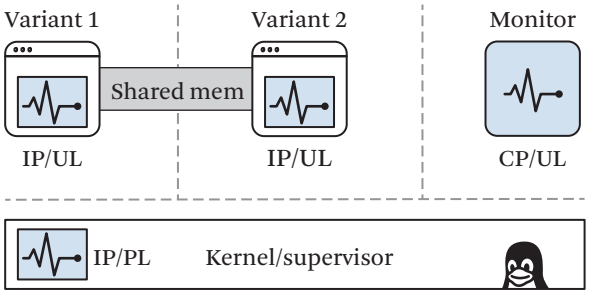


**Figure 8.1** Possible placements of an MVEE in the software stack.

architecture-specific. IP/PL monitors are therefore significantly harder to port to other platforms than CP monitors.

VARAN finally implements a third design that is represented by "IP/UL" in Figure 8.1 [Hosek and Cadar 2015]. VARAN is a reliability-oriented IP monitor, embedded into the variants. It consists of several components, each of which can communicate directly with the variant in which it is embedded. VARAN primarily intends to increase the reliability of software, e.g., by running two variants, one with and one without a new patch applied to them, to test that the patch does not introduce unintended side effects. It therefore uses a less secure design than the aforementioned "CP/UL" and "IP/PL" MVEEs. VARAN's authors also recognize this fact.

GHUMVEE is a security-oriented MVEE and is therefore implemented as a CP/UL MVEE. In Section 8.5 we also describe a hybrid design called ReMon [Volckaert et al. 2016]. ReMon is based on GHUMVEE, but it also includes an in-process component and a small kernel component, which makes ReMon a hybrid CP+IP/UL+PL MVEE.

### 8.1.3 Monitor-Variant and Monitor-Monitor Interaction

The MVEE's monitor and the variants interact whenever the variants trigger an event that is subject to monitoring. These events typically include executing a system call and raising a processor exception (e.g., by executing a privileged instruction or causing a segmentation fault). Each interaction requires transferring the control flow from the variant to the monitor and back, and may require copying the register context or memory contents of the variant to the monitor. Several mechanisms exist to fulfill each of these tasks. The placement of the monitor in the software stack defines which mechanisms are available.

#### 8.1.3.1 Control-Flow Transfer

The most trivial way to transfer control from the variant to the monitor and back is to invoke the monitor directly using a branch instruction. This is only possible for IP/UL monitors, which operate in the same address space and at the same privilege level as the variants. This control-flow transfer method is efficient but not very secure, as the variants typically invoke the monitor at their own discretion. Compromised variants could, for example, easily execute a system call without invoking the monitor first.

IP/PL monitors, which operate in the same address space but at a higher privilege level than the variants, cannot be invoked directly. Instead, these monitors

must be invoked by the kernel or supervisor's system call and trap handlers, either by patching these handlers or by installing hooks. When a variant executes a system call or triggers an exception, the processor transfers control to the kernel/supervisor's system call handler or exception handler, and the handler must then invoke the monitor. This interaction method is fully secure. Since the monitor invocation is handled by the kernel/supervisor, compromised variants are not able to escape the monitoring mechanism.

CP/UL monitors, which operate in a separate address space, cannot be invoked directly either. Instead, they rely on the operating system's debugging interface to "attach" to the variant, thus establishing a debugger-debuggee relationship between the monitor on one side and the variants on the other side. With such a relationship in place, the operating system will suspend the execution of the debuggee (variant) whenever it triggers an event that requires the attention of the the debugger (monitor). The operating system will then schedule the monitor and make information about the event available to the monitor. This interaction method is also fully secure but incurs significant runtime overhead, since synchronous interaction between two separate processes requires context and TLB flushes.

ReMon, the hybrid design we describe in Section 8.5, consists of both an IP monitor and a CP monitor. ReMon has a system call broker component that intercepts system calls in kernel space and invokes the appropriate monitor based on a user-defined policy. The system call broker can invoke either the CP monitor, using the operating system's debugging interface, or the IP monitor, by pointing the user-space program counter at the IP monitor's known entry point before exiting kernel space. ReMon's system call handling mechanism is fully secure and more efficient than the one used by CP/UL monitors.

#### 8.1.3.2 Register Context and Data Transfer

The MVEE's monitor requires access to the variant's register context, e.g., to read the system call number, and to the variant's virtual memory, e.g., to read system call arguments.

The variant's virtual memory can be accessed directly by all IP/UL and IP/PL monitors, as these monitors share their address space with the variant. IP/UL monitors also share their register contexts with the variant and can therefore access this context directly. IP/PL monitors do not share their register contexts with the variant. Instead, they must access a copy of this context. The processor stores this copy at a pre-defined location whenever it changes the privilege level. The performance overhead incurred by having to access the copy of the register context is negligible.

CP monitors can access neither the register context nor the variant's memory directly. Instead, they rely on the OS's debugging interfaces to transfer this information to the monitor's address space. Such transfers incur significant runtime overhead.

#### 8.1.3.3 Inter-monitor Communication

Some MVEE designs, particularly the IP/UL ones, use multiple monitor instances, each embedded into or assigned to just one variant. The instances frequently communicate with each other, e.g., to verify if all variants execute the same system call. While most operating systems offer a variety of options for inter-process communication, all MVEEs that fall into this category use a ring buffer backed by a shared memory region for inter-monitor communication.

#### 8.1.3.4 Performance Implications

Koning et al. [2016] conducted the most comprehensive study to compare monitor-variant interaction mechanisms. Through a series of micro-benchmarks, they showed that intercepting system calls and invoking the monitor in kernel space, similar to N-Variant Systems' IP/PL monitor [Cox et al. 2006], generally yields the highest performance. Using hardware virtualization features to intercept system calls and running the monitor in supervisor mode, similar to MvArmor's IP/PL monitor [Koning et al. 2016], is marginally faster than a kernel-based design if the monitor can emulate the system call, and up to 7.59× slower if the call cannot be emulated. Intercepting system calls using the OS's debugging interface, as is done in all existing CP/UL monitors, is up to two orders of magnitude slower than the mechanisms used in IP/PL monitors.

Prior to this study, Volckaert et al. [2013] compared data transfer mechanisms used in CP/UL monitors and showed that GHUMVEE's `ptrace` extension yields significantly faster monitor-variant data transfers than Orchestra's shared-memory-based mechanism [Salamat et al. 2009], while the latter, in turn, is significantly faster than regular `ptrace`-based data transfers.

## 8.2 Implementation of GHUMVEE

GHUMVEE is a CP/UL monitor and thus relies on the OS's debugging interface to set up and communicate with the variants. GHUMVEE launches the variants 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 variant 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 variants after they have raised this signal, and it reports their status to the monitor. The monitor can then resume the variants and begin to monitor their execution.

### 8.2.1 Monitoring System Calls

Like most MVEEs, GHUMVEE monitors the variants' behavior at the system call interface by intervening at the kernel level at the entry and exit of every system call. GHUMVEE leverages the operating system's debugging API to place the variants under the monitor's control, to intercept the variants' system calls, and to run the variants in *lockstep*. The monitor suspends each variant that enters or exits from a system call until all variants have reached the same entry or return point. When this happens, the variants are said to have reached a *RendezVous Point* (RVP) (sometimes referred to as a synchronization point).

The monitor asserts that the variants are in equivalent states whenever they reach such an RVP by comparing the system call arguments. Two sets of system call arguments are considered equivalent if they are identical (in the case of non-pointer arguments) or if the data they refer to is identical (in the case of pointer arguments). Salamat [2009] gives a formal definition of equivalent states.

If the variants are not in equivalent states at an RVP, the monitor raises an alarm and takes the appropriate action. GHUMVEE considers all tasks that share an address space with one of the variants that caused the discrepancy as tainted, and it therefore terminates these tasks. Do note that this does not necessarily stop the entire program. It is becoming a common practice to compartmentalize complex programs, such as web browsers and web servers, into multiple, mostly independent tasks that do not share address spaces. In some modern web browsers, for example, every open tab is backed by a separate process. Should GHUMVEE detect a discrepancy when running multiple variants of such a process, it would only terminate the browser tab that caused the discrepancy.

*Reliability-oriented* monitors that are, e.g., used to test new software patches may differ from *security-oriented* monitors, such as GHUMVEE, with respect to system call monitoring. For example, VARAN does not enforce lockstep execution [Hosek and Cadar 2015]. Instead, it lets the master variant run ahead of the slave variants and caches the arguments and results of all the master variant's system calls so that they may be consulted by the slave variants at a later point.

### 8.2.2 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 runtime overhead,

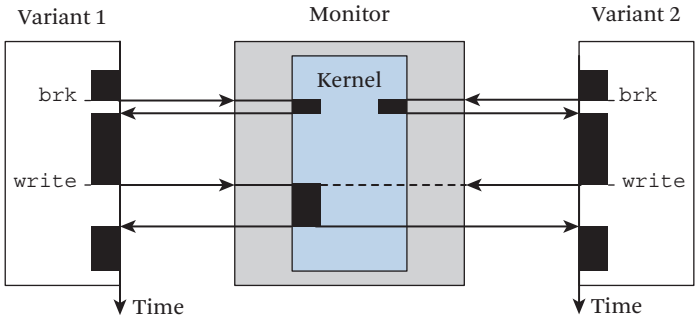


**Figure 8.2** Transparently executing I/O-related system calls.

the end user should not be able to notice that more than one variant of the program is running. GHUMVEE therefore uses a master/slave replication model. One of the variants is the designated master variant and the other variants are slaves. GHUMVEE ensures that only the master variant 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 variants reach an RVP at the start of an I/O-related system call, GHUMVEE verifies that the variants are in equivalent states, and then overwrites the system call number in the slave variants with that of a system call with no visible effects. GHUMVEE currently uses `sys_getpid` for this purpose since it is a trivial and fast system call. When GHUMVEE subsequently resumes all variants, only the master variant executes the intended I/O operation.

At the next RVP, when all variants have returned from their system call, GHUMVEE copies the results of the system call from the address space of the master to the address space of the slave variants. 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 variants are called *normal calls*. In Figure 8.2, the handling of the normal call `brk` is shown, as well as that of the master call `write`.

### 8.2.3 Injecting System Calls and Restarting Variants

On top of the above tasks, GHUMVEE can also inject new system calls and, as a result, rewind variants to their initial state. Injecting system calls can be useful to add new functionality to the variants transparently. To inject a system call in a variant, GHUMVEE waits until the variant has reached an RVP. At this point,

GHUMVEE stores a backup of the register context of the variant and overwrites 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 variant that is large enough for the arguments. If the variant is multi-threaded, GHUMVEE searches for the variant's thread-local stack, in order not to corrupt memory that might be used by other tasks that share an address space with the variant.

GHUMVEE then reads and stores the original content of the memory page and writes the arguments into that page; it then resumes the variant and waits until the injected system call returns. At that point, GHUMVEE restores the original contents of the overwritten memory page and restores the original register context, prior to the system call injection.

Restarting variants 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 variant as well as the environment variables [GNU.org 2017] at the time of the original `sys_execve` invocation. Whenever a variant reaches an 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 variant. GHUMVEE uses this restart mechanism to enforce disjoint code layouts, as we will explain in Section 8.4.

## 8.3 Inconsistencies and False Positive Detections

MVEEs must feed all variants the same input in order to guarantee that they 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 8.2.2 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 variants often behave differently after reading input from such sources. This can lead to false positive detections by the monitor. In this section, we summarize the sources of input that can be accessed directly and describe how we provide consistent input from such sources to all variants.

### 8.3.1 Shared Memory

All commodity operating system kernels offer a file-mapping API and an Inter-Process Communication (IPC) API to share physical memory pages among multiple processes.

The file-mapping API, which can be accessed through the `sys_mmap` system call on Linux systems, allows 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 loads 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 denies any requests made by a non-privileged user to open the file with read/write access and instead allows only 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. Variants can read from shared memory pages without invoking a system call and, consequently, are not subject to the lockstep execution mechanism we discussed in Section 8.2.1 when doing so. The MVEE's monitor therefore cannot guarantee that the variants will read the same input from shared memory pages that are being written to by an external process. Similarly, the variants could also write to the pages directly, which prevents the MVEE's monitor from asserting that the variants write the same data to the pages.

A possible solution to this problem is to revoke the variants' 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 variants. 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 [Maebe et al. 2003] 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 have content that is completely static (i.e., the pages cannot be written to by either the variants or any external process that runs at the same privilege level). The monitor can therefore still guarantee that the variants will receive the same input. Allowing read-only shared mappings is necessary to support dynamically linked programs since the program interpreter's preferred method of loading shared libraries is by mapping them using the file-mapping API.[1]

GHUMVEE does not allow read/write shared mappings. The monitor generally returns an `EPERM` error when a variant attempts to establish such a mapping, thus indicating that the mapping is not allowed. In specific cases, however, read/write shared mappings are used not 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 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 a variant attempts to write to it for the first time. From that point onward, external processes can no longer influence the contents of

1. The program interpreter is a user-space OS component responsible for loading programs and setting up their initial virtual address space.

the privately mapped page, which eliminates the need for the monitor to replicate the contents of the pages to all variants. The monitor does, however, still verify whether the variants 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, the monitor raise an alarm. If they do match, however, the monitor writes the contents back to the backing file.

GHUMVEE's handling of shared memory is similar to Cavallaro's MVEE [Cavallaro 2007] but is more advanced than other security-oriented MVEEs because those do not support the mapping-type-override method.

### 8.3.2 Timing Information

Interactive and real-time applications frequently need to measure the length of a time interval to guarantee that they function correctly. Media players, for example, need to know exactly when to start rendering a frame. For such applications, the timing information must be accurate, precise, and 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 support the 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 [Intel 2014]. 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) [Linux Programmer's Manual 2017a]. 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. Unlike 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 cannot guarantee that the variants that access this information receive consistent input.

GHUMVEE implements work-arounds 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 variant [Intel 2014]. Setting this flag discards the variants' privileges to execute the `RDTSC` instruction. Whenever the variants try to execute an `RDTSC` instruction, the processor raises a general protection fault. The operating system translates this fault into a `SIGSEGV` signal and notifies the monitor accordingly. Whenever the monitor receives such a notification, it disassembles the instruction that caused the fault. If the instruction is indeed an `RDTSC`, GHUMVEE executes the instruction on the variants' behalf and replicates the results.

To eliminate the inconsistencies caused by the VDSO, GHUMVEE overrides 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 call the GHUMVEE Program Loader (GPL),[2] deletes the ELF auxiliary vector entry argument that specifies the location of the VDSO [Linux Programmer's Manual 2017b]. Afterward, GPL manually maps the original program into the virtual address space, sets up the initial stack exactly as it would have been set up had GHUMVEE not overridden the arguments of the `sys_execve` call, and passes the control to the original program. A program never invokes the VDSO directly but instead uses the wrappers provided by the C standard library (libc). If the ELF auxiliary vector entry for the VDSO is missing, however, libc falls back to using the original system call that each virtual system call corresponds to. These original system calls are intercepted by GHUMVEE's monitor. An alternative solution could be to replace the VDSO with a custom library that leverages GHUMVEE's USRVP replication infrastructure (see Section 8.3.7) to replicate the master variant's system call results to all slave variants.

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 [Hosek and Cadar 2015]. Our proposed solutions have a minimal performance impact on the many applications we tested. The `RDTSC` instruction is typically only used during the start-up and shutdown of a program, e.g., to measure the runtime of individual threads. Our proposed solution for the VDSO does significantly impact the latency on executing individual timing-related system calls. In Section 8.5, we propose a new monitor design that reduces this impact to a bare minimum.

2. GPL is available under the BSD 3-clause license at http://github.com/stijn-volckaert/ReMon/tree/master/MVEE_LD_Loader.

### 8.3.3 File Operations

Multi-variant execution should be transparent to the variants and to external observers. GHUMVEE therefore uses the master call mechanism to ensure that I/O operations are only performed once. With this mechanism, only the master variant performs the actual I/O operations, and the monitor replicates the results to the slave variants. Intuitively, it might also make sense to use master calls for system calls that open, modify, or close file descriptors. Regular files, however, might be mapped into the variants' address spaces using the file-mapping API. If we apply the master call mechanism to such files, any subsequent file-mapping request would fail in all slave variants. GHUMVEE therefore allows slave variants 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 variant and some are opened in all variants, the same file descriptor might have different values in the different variants. As GHUMVEE must ensure that the multi-variant execution is transparent to the variants, the monitor replicates the same file descriptor values to all variants, regardless of whether or not they have actually opened the file. Whenever the variants perform a normal system call that they must all execute, GHUMVEE maps the replicated file descriptor value back to the original file descriptor value at the system call entrance site. When the call returns, GHUMVEE maps the original file descriptor value back to the replicated file descriptor value.

Although few 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 most I/O operations on behalf of the variants. Variants running in Orchestra therefore do not open any file descriptors other than those for regular files.

### 8.3.4 Signal Handling

UNIX systems use signals as a general-purpose mechanism to send processes notifications [Linux Programmer's Manual 2017c]. Each notification has a signal number associated with it, and the signal number generally defines the notification's meaning. For example, when a program performs an invalid memory access, the kernel sends it a `SIGSEGV` signal.

Broadly speaking, 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, the signal is 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.

Supporting control-flow signals in an MVEE is generally straightforward, as they occur at the same point in each variant's execution. Supporting asynchronous signals sent to the variants is extremely challenging, however. These signals can easily trigger behavioral divergences in the variants if their delivery is not meticulously controlled by the MVEE's monitor. Since the monitor generally only intervenes in the variants when they execute a system call, the variants may execute freely for the most part. Consequently, one variant can easily run ahead of the others, and they are generally not in equivalent states until they reach an RVP.

If the behavior of a signal handler used to handle an asynchronous signal depends on the state of the variant in any way, delivering these asynchronous signals directly all but guarantees behavioral divergence and thus a false positive alarm. MVEEs that support asynchronous signals therefore attempt to defer their delivery until the variants reach an RVP. At an RVP, the variants are in equivalent states, and the asynchronous signals can be delivered synchronously without inducing a divergence.

While the general principle of deferred synchronous signal delivery is simple, its implementation is not. To the best of our knowledge, GHUMVEE is the only MVEE that overcomes all the intricacies of asynchronous signal delivery and implements deferred synchronous signal delivery correctly. We refer to Volckaert's PhD dissertation for a full overview of challenges that need to be overcome when implementing deferred signal delivery in an MVEE [Volckaert 2015]. These challenges include, but are not limited to, correct handling of blocked and ignored signals, support for per-thread signal masks, support for system call interruption, support for master call interruption, and support for `sys_sigsuspend` and `sys_rt_sigsuspend`.

GHUMVEE's mechanism for handling asynchronous signal delivery is optimized for correctness rather than performance. During our evaluation, we concluded that almost every program that relies on signal handling still functions correctly inside GHUMVEE. The only exception is the `john-the-ripper` program in the `phoronix` 4.8.3 benchmark suite. This program waits for signals to be de-

livered in a busy loop, in which no system calls are used. Therefore, if GHUMVEE intercepts a signal that is delivered to the variants, it indefinitely defers the delivery of the signal because the variants 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 [Salamat et al. 2009]. Orchestra uses a heuristic to determine whether a signal can be safely delivered, even if its variants have not reached a system call RVP yet. However, Orchestra does not handle signals that interrupt system calls correctly.

VARAN's signal-handling mechanism is ideal with respect to performance and correctness [Hosek and Cadar 2015]. VARAN is an IP monitor and therefore does not rely on the `ptrace` API. Furthermore, VARAN forces its follower variants not to invoke system calls at all. Instead, the follower variants just wait for the results of the leader variants' system calls. In VARAN, the leader variant accepts and processes incoming signals without delay. While the follower variants generally do not receive signals at all, the leader variant logs the metadata associated with the signal into the event streaming buffer. This metadata provides the follower variants with sufficient information to replay the invocation of the signal handler truthfully.

### 8.3.5 Address-Sensitive Behavior

Most of the sources of inconsistencies in the behavior of single-threaded variants can be eliminated or mitigated by the monitor itself. The one notable exception is *address sensitivity*, a problem frequently encountered in real-world software. The monitored 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 variants can therefore lead to false positive detections by the monitor. We have identified three problematic idioms that lead to address sensitivity, and we discuss them now.

**Address-Sensitive Data Structures.** We have frequently encountered programs that use data structures whose runtime layout and shape 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: the (hash) keys used to select buckets and to order elements are

based on the numerical pointer values. The problem in general is that the address-sensitive behavior induces changes in the shape of allocated data structures, i.e., in the way they are linked via pointer chains.

Applying diversification techniques that result in diversified address space layouts and shapes eventually yields divergences in the variants' system call and synchronization behavior. For example, in address-sensitive hash tables an insertion of the same object can trigger a hash table collision and a subsequent memory allocation request (e.g., through a `sys_mmap` call) to resize the table in some variants 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 variants, 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 variants: tolerating a minor discrepancy early on leads to bigger and bigger discrepancies in the synchronization behavior and, consequently, in the system call of the variants, to the point where we can no longer distinguish between benign discrepancies and compromised variants.

Dynamic memory allocators are the instigators of this ripple effect. For example, GNU libc's `ptmalloc` 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 discuss in the next section, GHUMVEE replicates the master variant's synchronization operations in the slave variants. Thus, if the variants behave differently with respect to memory allocations, the replicated synchronization information might be misinterpreted by other variants because it does not match their actual behavior. From that point onward, such variants 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 identified in `ptmalloc` is its requirement that any memory region it allocates through `sys_mmap` is aligned to a large boundary of, e.g., 1 MiB. The operating system only guarantees that newly allocated memory regions are aligned to a boundary equal to the size of a physical memory page. To bridge this gap, `ptmalloc` always allocates twice the memory it needs and subsequently deallocates the region before and the region after the boundary. When running multiple variants 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` therefore only deallocates the upper region. This might trigger false positive detections in MVEEs that execute their variants in lockstep, since some variants 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 leads to only 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 and the compiler are not to blame. An optimizing compiler aligns members of a data structure to their natural boundary. If necessary, padding bytes are inserted between the members. These padding bytes are never used, and it is therefore acceptable that they are not initialized. If that is the case, and they 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 variants' 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 argue in the next section.

### 8.3.6 Nondeterminism in Parallel Programs

Except for the few cases we discussed in the previous sections, single-threaded variants produce the same outputs when given the same program inputs. The same is not true of multi-threaded variants, in which threads may communicate directly through shared memory, without using system calls. In these variants, the output also depends on the runtime thread interleaving. Security-oriented MVEEs, which run variants in strict lockstep, must therefore control the thread interleaving such that each variant makes the same system calls with equivalent arguments.

Two lines of research address exactly this challenge. On the one hand, there are the Deterministic Multi-Threading (DMT) systems, which repeat the same thread interleaving when given the same program inputs [Basile et al. 2002, Reiser et al. 2006, Berger et al. 2009, Liu et al. 2011, Merrifield and Eriksson 2013, Cui et al. 2013, Olszewski et al. 2009, Lu et al. 2014, Devietti et al. 2009, Bergan et al. 2010, Zhou et al. 2012]. On the other hand, there are online Record/Replay (R/R) systems, which capture the thread interleaving in one running instance of a program and impose this captured schedule in a concurrently running instance [Basile et al. 2006, Lee et al. 2010, Basu et al. 2011].

DMT systems are an ill fit in the context of MVEEs, as such systems are likely to impose a different thread interleaving whenever the program code or code layout changes. Thus, running diversified variants in which the code layout differs with near certainty and with DMT on top of an MVEE will still result in different thread interleavings and, consequently, divergent behavior. We refer to our earlier work for an in-depth reasoning to support this argument [Volckaert et al. 2017].

In GHUMVEE, we therefore opt for the second option. R/R systems usually capture the thread interleaving at the granularity of synchronization operations (e.g., pthread mutex operations). The underlying thought is that in data-race-free programs, any inter-thread communication must, by definition, be protected by critical sections. Imposing an equivalent synchronization operation schedule in every execution of the program thus trivially leads to a thread interleaving that is equivalent for each run. Since synchronization operations are not likely to differ between diversified variants, R/R systems are a much better fit than DMT in the context of an MVEE.

### 8.3.7 User-Space Rendezvous Points

In order to maintain equivalent system call behavior, even in parallel programs or in programs that feature address-sensitive behavior, we introduce *user-space rendezvous points* (USRVPs). Conceptually, we add these USRVPs to any operation in the variants' code if (i) the operation may affect the variants' system call behavior and (ii) the operation may produce different results in each variant. At each USRVP, we insert calls to a replication agent. This replication agent is a shared library we forcefully load into each variant's address space. As shown in Figure 8.3, the replication agent captures the results of the instrumented operation in the master variant and stores them in a circular buffer that is visible to all variants. The agent then forces the slave variants to overwrite the results of their own instrumented operations with the results produced by the master variant.

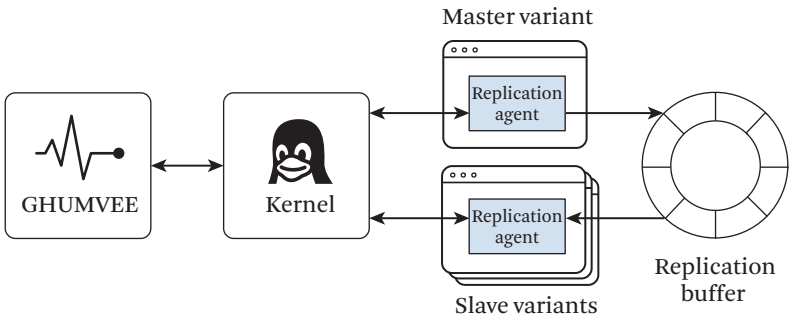


**Figure 8.3** Using replication agents to replicate nondeterministic program behavior.

We developed three components that help a developer with the implementation of USRVPs and replication agents:[3]

**The GHUMVEE replication API.** The GHUMVEE replication API can be used to generate the replication agents and the USRVPs. 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 variant, the generated function can retrieve the results of the instrumented operation by calling a programmer-specified function. It can then record the input into a circular buffer. In the slave variants, the generated function retrieves the results from the circular buffer. The replication API further allows the programmer to specify whether or not the slaves should also execute the instrumented operation. This may be necessary, e.g., if the instrumented operation has side effects that may affect future system call and synchronization behavior.

**The Lazy hooker.** The generated USRVP functions can be embedded in the variant by registering them with a shared library, which we call "the lazy hooker." This library monitors the dynamic loading process of the variants and determines whether or not a USRVP generated with the above API should be installed. At the time of the registration, the lazy hooker may insert the USRVP function immediately, if the specified library has already been loaded, or it can defer the insertion until the program loads the library.

3. We refer interested readers to Volckaert's Ph.D. thesis for an extensive discussion that includes usage examples [Volckaert 2015].

**The LinuxDetours library.** We insert the USRVP functions using `LinuxDetours`, a runtime 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 [Hunt and Brubacher 1999]. `LinuxDetours` can redirect calls to functions and generate trampolines that may be used to call the original function, without interception.

#### 8.3.7.1 USRVP Applications

GHUMVEE currently relies on USRVPs for two purposes. First, we add USRVPs to all thread synchronization operations in order to embed our R/R system into the variants. Thanks to our replication APIs and infrastructure, we were able to construct an R/R system that captures the order of thread synchronization operations in the master variant and replays an equivalent order in the slave variants. Contrary to most existing R/R systems, which capture only high-level synchronization operations, such as `pthread` mutex operations, we capture the order of *all* thread synchronization operations, including individual atomic instructions.

Second, we add USRVPs to operations on address-dependent data structures, such as the ones described in Section 8.3.5. At these USRVPs, we capture values such as pointer hashes and pointer comparison results used by sorting algorithms, and we force all the variants to use the same values. Thanks to our USRVPs, GHUMVEE can ensure that address-dependent data structures potentially allocated at different memory locations have the same shape in all variants.

It is important to note that USRVPs are application specific. Although we believe that the identification of USRVP insertion points can be automated to some extent, some help from the application developer will always be required. We refer to our earlier work for details on USRVP insertion point identification, the construction of efficient replication agents and R/R systems, and automation opportunities [Volckaert et al. 2013, Volckaert et al. 2017].

#### Evaluation and Comparison with other MVEEs

We applied our replication API and infrastructure to run a variety of applications, including the SPLASH-2x and PARSEC 2.1 parallel benchmark suites and two popular, though now outdated, desktop programs: the Firefox 3.6 browser and the LibreOffice 4.5 office suite.

To run the parallel benchmark suites, we constructed a replication agent that contains an R/R system and embedded this agent into `glibc`. This R/R replication agent is called from the more than 1,000 USRVPs we added to thread synchroniza-

tion operations in the `glibc`, `libgomp`, `libpthread`, and `libstdc++` libraries, as well as a few of the program binaries.

To run Firefox and LibreOffice, we constructed five different replication agents, each one supporting a particular address-sensitive data structure.

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 and user-space synchronization operations.

## 8.4 Comprehensive Protection against Code-Reuse Attacks

In 2007 Shacham presented the first Return-Oriented Programming (ROP) attacks for the x86 architecture [Shacham 2007]. 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 architectures such as SPARC [Buchanan et al. 2008], ARM [Kornau 2010], and many others.

### 8.4.1 Disjoint Code Layouts

As an alternative protection against user-space ROP attacks, we present Disjoint Code Layouts (DCL). With this diversification technique, GHUMVEE ensures 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 consistently. Our design and implementation of DCL offers many advantages over existing solutions:

- DCL offers complete immunity against user-space ROP attacks rather than just raising the bar for attackers.
- The execution slowdown incurred by this form of diversification is minimal.
- 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 protect against other types of exploits as well.
- DCL is compatible with existing compilers and existing solutions such as stack canaries [Cowan et al. 1998] and control-flow integrity [Abadi et al. 2005a, Tice et al. 2014].
- Unlike some existing techniques for memory layout diversification in MVEEs, 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. Systemwide, DCL does of course still cause considerable memory overhead due to its duplication of process-local data regions, such as the heap and writable pages. Still, GHUMVEE with DCL outperforms memory-checking tools in this regard.

Our technique of 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` requests 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. (The few exceptions are discussed later.)
- 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 variant enters a system call, this constitutes an RVP for GHUMVEE, at which point 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 variant'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 variants 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.

### 8.4.2 Mapping Segments Normally Mapped by the Kernel

Some code regions require special handling, however. Under normal circumstances the kernel maps the program image (i.e., the main binary's segments), the program interpreter, and the VDSO. When ASLR is enabled, it maps them at randomized addresses. But randomized addresses in all the variants do not suffice

to guarantee disjoint code layout. Because GHUMVEE cannot intervene in decision processes in kernel space, it therefore needs to prevent the kernel from mapping them and instead have them mapped from user space, i.e., by the program interpreter. GHUMVEE can then again intercept the mapping system calls and enforce non-overlapping mappings of code regions.

**Disjoint Program Images.** The standard way to launch new applications is to fork off a running process and 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 mentioned components and a main process stack into its now empty address space.

Mapping the program image from within user space is rather 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. In order to allow GHUMVEE to choose a different address for the program image in each variant, the program needs be compiled into a Position-Independent Executable (PIE). Recent versions of GCC and LLVM can do so without introducing significant overheads.

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 them to the kernel. This way, GHUMVEE can control the layout of the variants' processes it spawns itself as well as the layout of all the processes subsequently spawned within the variants. 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.

**Program Interpreter.** Even with the above indirect program invocation, we cannot prevent the kernel itself from mapping the program interpreter. Hence the indirect invocation does not suffice to ensure that no code regions overlap in the variants.

In Linux, the interpreter is only mapped when the kernel loads a dynamically linked program. To prevent that loading even when launching dynamically linked programs, we developed a statically linked loader program, hereafter referred to as the GHUMVEE Program Loader (GPL). Whenever an application is launched under the control of GHUMVEE, it is launched by launching GPL and having GPL load the actual application. Launching GPL is again done by intercepting the original `sys_execve` calls in GHUMVEE and rewriting their arguments.

**VDSO.** In each variant launched by GHUMVEE, the copy of GPL is started under GHUMVEE's control. At GPL's entry point, GHUMVEE first checks whether the VDSOs, which were allocated randomly in each variant with ASLR, are disjoint. If they are not, GHUMVEE restarts new variants until a layout is obtained in which the VDSOs are disjoint. Until recently, the Linux kernel mapped the VDSO anywhere between 1 and 1,023 pages below the stack on the i386 platform. It was therefore not uncommon that GHUMVEE had to restart one or more variants. 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, GPL manually maps the program interpreter through `sys_mmap2` calls. This way, GHUMVEE can override the base addresses of the variants' interpreters to map them onto regions that contain no code in the other variants.

**Program Stack.** Next, GPL sets up an initial stack in each variant 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 GPL itself, but this is rather simple to implement.

GPL then transfers control to GHUMVEE through a pseudo system call. GHUMVEE intercepts this call and modifies the call number and arguments such that the kernel unmaps GPL in each variant. Upon return from the call to GHUMVEE, it transfers control to the program interpreter. When the variants then resume, they have fully disjoint code layouts.

**Original Program and Shared Libraries.** In each variant, the interpreter then continues to load and map the original program and the shared libraries, all of which will be subject to DCL as GHUMVEE intercepts the invoked system calls. Afterward, the

interpreter passes control to the program in each of the variants, which are then all ready to start executing the actual programs.

Assuming that the original program stack is protected by W⊕X, the summarized loading and mapping process 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 is two or more variants with completely disjoint code regions. Any divergence in I/O behavior caused by a ROP attack successfully attacking one variant will be detected and aborted by the monitor.

### 8.4.3 Disjoint Code Layout vs. Address Space Partitioning

Cox et al. and Cavallaro independently proposed to combat memory exploits with essentially identical techniques they called Address Space Partitioning (ASP) [Cox et al. 2006] and Non-Overlapping Address Spaces [Cavallaro 2007], respectively. We will refer to both as ASP.

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

Another significant difference between both of the proposed ASP techniques and DCL is that both implementations of ASP require modifications to either the kernel or the program loader. Cox's 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 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 is the case with the relaxed monitoring policies we describe in Section 8.5, by refraining from modifying core system libraries, GHUMVEE offers the end user a great degree of flexibility in when, how, and where its security features should be used.

Finally, whereas DCL relies on PIE [Murphy 2012] to achieve non-overlapping code regions in the variants, both presented forms of ASP rely on standard, non-PIE ELF binaries, despite the fact that PIE support was added to the GCC/binutils toolchain 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 facilitate widespread adoption of DCL.

### 8.4.4 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 variants 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, the memory page was not subject to DCL at the time it was mapped and GHUMVEE therefore refuses 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 inevitably prevents 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 does 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 do 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 [Moser 2006, Drepper 2006]. 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 8.2, GHUMVEE does not allow the

use of shared memory. Applications that use the double-mmap method therefore would 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. The double-mmap 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.

### 8.4.5 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 has its protections set to `PROT_EXEC` in more than one variant. Furthermore, GHUMVEE's replication ensures that all variants receive exactly the same input. This is the case for input provided through system calls and through signals. Combined, these 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 is executable in no more than one variant. The operating system's memory protection makes the variants crash as soon as they try to execute code in their non-executable or missing page at the same virtual address.

We should point out, however, that 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 runtime intrusion prevention evaluator (RIPE) [Wilander et al. 2011], are not necessarily 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 variants, rather than being replicated and intercepted by the replication engine.

To validate the above claimed effectiveness of GHUMVEE with DCL to some extent, we constructed four ROP attacks against high-profile targets. The attacks are available at http://github.com/stijn-volckaert/ReMon/.

Our first attack is based on the Braille tool by Bittau et al. [2014]. 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 when nginx is run inside GHUMVEE with only one variant. If we run the attack on two variants, however, it fails to leak the stack canary. While attempting to leak the stack canary, at least one variant crashes for every attempt. Whenever a variant crashes, GHUMVEE assumes that the program is under attack and shuts down all other variants 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 with DCL blocks this attack, the attack probably would not have worked even with DCL disabled: with more than one variant, the attack's stack-reading step can only succeed if every variant 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 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 (CVE-2010-4221) in the `proftpd` FTP server. The attack scans the `proftpd` binary and the `libc` library for gadgets for 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 one variant. When run with two variants, GHUMVEE detects that one variant 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 variants in the same logical process. During the attack, proftpd's master process manages to restart worker processes quickly enough to keep the server available throughout the attack.

Our third attack exploits a stack-based buffer overflow (CVE-2012-4409) in `mcrypt`, an encryption program intended to replace `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 one variant. When run with two variants, GHUMVEE detects a crash in one variant 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 one variant. When run with two variants, GHUMVEE detects a crash in one variant and an attempt to perform a system call in the other. It therefore shuts down the program to prevent any damage to the system.

## 8.5 Relaxed Monitoring

Most of the security-oriented MVEEs that preceded GHUMVEE incur non-negligible runtime performance overhead. This overhead can be attributed to two key design decisions: the strict lockstep synchronization model for system calls and the CP/UL operation of the MVEE's monitor.

Both of these design decisions aggressively favor security over performance. In this section, we revisit these key decisions and present a new MVEE design called ReMon. ReMon incurs significantly lower runtime overhead than other CP/UL MVEEs, while maintaining a high level of security.

Our design is motivated by the fact that a security policy of monitoring *all* system calls is overly conservative [Garfinkel et al. 2004, Provos 2003]. Many system calls cannot affect any state outside of the process making the system call. Only a small set of *sensitive* system calls are potentially useful to an attacker. Thanks to its IP-MON component discussed below, ReMon supports configurable *relaxation* policies that allow non-sensitive calls to execute without being cross-checked against other variants.

### 8.5.1 ReMon Design and Implementation

Like GHUMVEE, ReMon supervises the execution of multiple diversified program variants that run in parallel. ReMon's main goals are (i) to monitor all security-sensitive system calls—hereafter referred to as "monitored calls"—issued by these variants; (ii) to force monitored calls to execute in lockstep; (iii) to disable monitoring and lockstepping for non-security-sensitive system calls—hereafter referred to as "unmonitored calls"—thus allowing the variants to execute these calls as efficiently as possible while still providing them with consistent system call results; and (iv) to support configurable monitoring relaxation policies that define which

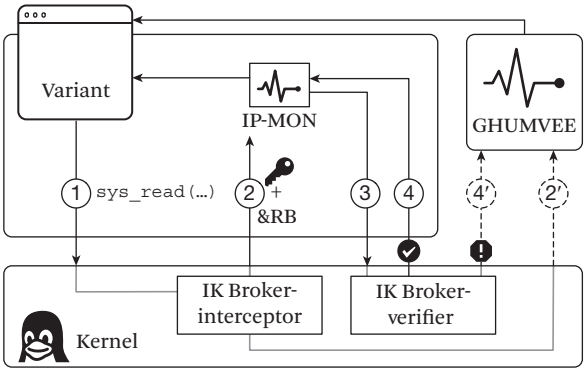


**Figure 8.4** ReMon's major components and interactions.

subset of all system calls is considered non-security-sensitive, and therefore should not be monitored. ReMon uses three main components to attain these goals:

**GHUMVEE** is the security-oriented CP monitor implemented as discussed in the preceding sections. Although GHUMVEE can be used in stand-alone fashion, it only handles monitored calls when used as part of ReMon.

**IP-MON** is an in-process monitor loaded into each variant as a shared library. IP-MON provides the application with the necessary functionality to replicate the results of unmonitored calls.

**IK-B** is a small in-kernel broker that forwards unmonitored calls to IP-MON and monitored calls to GHUMVEE. IK-B also enforces security restrictions on IP-MON and provides auxiliary functionality that cannot be implemented in user space. The broker is aware of the system calls that IP-MON handles and of the relaxation policy that is in effect.

These three components interact whenever a variant executes a system call, as shown in Figure 8.4. Our kernel-space system call broker, IK-B, intercepts the system call ① and either forwards it to IP-MON ② or to GHUMVEE ②'. The call is forwarded to IP-MON only if the variant has loaded an IP-MON that can replicate the results of the call, and if the active relaxation policy allows the invoked call to be executed as an unmonitored call. If these two criteria are not met, IK-B uses the standard `ptrace` facilities to forward the call to GHUMVEE instead, which handles it exactly as a regular CP-MVEE.

In the former case, IK-B forwards the call by overwriting the program counter so that the system call returns to a known "system call entry point" in IP-MON's

executable code. While doing so, IK-B gives IP-MON a one-time authorization to complete the execution of the call without having the call reported to GHUMVEE. The broker grants this authorization by passing a random 64-bit token ② as an implicit argument to the forwarded call. IP-MON then performs a series of security checks and eventually completes the execution of the forwarded call by restarting it ③. IP-MON can choose to restart the call with or without the authorization token still intact. If the token is intact upon reentering the kernel, IK-B allows the execution of the system call to complete and returns the call's results to IP-MON ④. If the token is not intact, or if IP-MON executes a different system call, or if the first system call executed after a token has been granted does not originate from within IP-MON itself, IK-B revokes the token and forces the call to be forwarded to GHUMVEE ④.

IP-MON generally executes unmonitored system calls only in the master variant and replicates the results of the system call to the slave variants through the Replication Buffer (RB) discussed in Section 8.5.1.2. The slaves wait for the master to complete its system call and copy the replicated results from the RB when they become available.

Although IP-MON allows the master variant to run ahead of the slaves, it still checks if the variants have diverged. To do so, the master's IP-MON deep-copies all its system call arguments into the RB, and the slaves' IP-MONs compare their own arguments with the recorded ones when they invoke IP-MON. This measure minimizes opportunities for asymmetrical attacks (cf. Section 8.5.2).

#### 8.5.1.1 Securing the Design

The IK-B verifier only allows variants to complete the execution of unmonitored system calls if those calls originate from within an IP-MON instance having a valid one-time authorization token. As only the IK-B interceptor can generate valid tokens, this mechanism *forces every unmonitored system call to go through IK-B.* At the same time, it also ensures that IP-MON can *only execute unmonitored system calls if it is invoked by IK-B and it is invoked through its intended entry point*. This mechanism is, in essence, a form of control-flow integrity [Abadi et al. 2005a]. It also allows us to hide the location of the RB, thereby preventing the RB from being accessed from outside IP-MON. Protecting the RB is of critical importance to the security of our MVEE, as we discuss in Section 8.5.2. To fully hide the location of the RB while still allowing benign accesses, we ensure that the pointer to the RB is only stored in kernel memory.

IK-B loads the RB pointer and the token into designated processor registers whenever it forwards a call to IP-MON, and IP-MON is designed and implemented

such that it does not leak these sensitive values into user-space-accessible memory. First, we compile IP-MON using `gcc` and use the `-ffixed-reg` option to remove the RB pointer and authorization token's designated registers from `gcc`'s register allocator. This ensures that the sensitive values never leak to the stack nor to any other register. Second, we carefully craft specialized accessor functions to access the RB. These functions may temporarily load the RB pointer into other registers, e.g., to calculate a pointer to a specific element in the RB, but they restore these registers to their former values upon returning. We also force IP-MON to destroy the RB pointer and authorization token registers themselves upon returning to the system call site. Finally, we use inlining to avoid indirect control-flow instructions from IP-MON's system call entry point. This ensures that IP-MON's control flow cannot be diverted to a malicious function that could leak the RB pointer or authorization token.

ReMon further prevents discovery of the RB through the /proc/maps interface: it forcibly forwards all system calls accessing the maps file to GHUMVEE and it filters the data read from the file. This requires marking the maps file as a special file, as described in Section 8.5.1.6.

To prevent IP-MON itself from being tampered with, we also force all system calls that could adversely affect IP-MON to be forwarded to GHUMVEE. These calls (e.g., `sys_mprotect` and `sys_mremap`) are then subject to the default lockstep synchronization mechanism.

#### 8.5.1.2 The IP-MON Replication Buffer

Like the replication agents discussed in Section 8.3.7, IP-MON must be embedded into all variants, so it consists of multiple independent copies, one per variant. These copies must cooperate, which requires an efficient communication channel. Although a socket or FIFO could be used, we opted for an RB stored in a memory segment and shared by all the variants.

To increase the scalability of our design, we opted not to use a true circular buffer. Instead, we use a linear RB. When our RB overflows, we signal GHUMVEE using a system call. GHUMVEE then waits for all variants to synchronize, resets the buffer to its initial state, and resumes the variants. Involving GHUMVEE as an arbiter avoids costly read/write sharing on RB variables that keep track of where data starts and ends in the RB. Instead, each variant thread only reads and writes its own RB position. The implementation of the IP-MON RB is nearly identical to that of the RBs GHUMVEE uses to support USRVPs (see Section 8.3.7).

```
/* read(int fd, void * buf, size_t count) */
MAYBE_CHECKED(read) {
  // check if 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 args each variant passed to the call.
  // if they match, we allow only the master to complete the call,
  // while the slaves wait for the master's results.
  CHECKREG(ARG1);
  CHECKPOINTER(ARG2);
  CHECKREG(ARG3);
  return MASTERCALL | MAYBE_BLOCKING(ARG1);
}

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

**Listing 8.1** Replicating the read system call in IP-MON.

#### 8.5.1.3 Adding System Call Support

ReMon currently supports well over 200 system calls. To provide a fast path, IP-MON supports a subset of 67 system calls. However, adding support to IP-MON for a new system call is generally straightforward. IP-MON offers a set of C macros to describe how to handle the replication of the system call and its results.

As an example, Listing 8.1 shows IP-MON's code for the `read` system call. The code is split across four handler functions that each implement one step in the handling of a system call using the C macros provided by IP-MON.

First, the `MAYBE_CHECKED` function is called to determine if the call should be monitored by GHUMVEE. If the `MAYBE_CHECKED` handler returns `true`, IP-MON

forces the original system call to be forwarded to GHUMVEE (④) by destroying the authorization token and restarting the call.

IP-MON uses a fixed-size RB to replicate system call arguments, results, and other system call metadata. Prior to restarting the forwarded call, we therefore need to calculate the maximum size this information may occupy in the RB. If the size of the data as calculated by the CALCSIZE handler exceeds the size of the RB, IP-MON forces the original system call to be forwarded to GHUMVEE. If the data size does not exceed the size of the RB, but it is bigger than the available portion of the RB, the master waits for the slaves to consume the data already in the RB, after which it resets the RB.

Next, if IP-MON has decided not to forward the original system call to GHUMVEE, it calls the PRECALL handler. In the context of the master variant, this function logs the forwarded call's arguments, call number, and a small amount of metadata into the RB. This metadata consists of a set of boolean flags that indicate whether or not the master has forwarded the call to GHUMVEE, whether or not the call is expected to block when it is resumed, etc. If the function is called in a slave variant's context, IP-MON performs sanity checking by comparing the slave's arguments with the master's arguments. If they do not match, IP-MON triggers an intentional crash, thereby signaling GHUMVEE through the `ptrace` mechanism and causing a shutdown of the MVEE.

The return value of the PRECALL handler determines whether the original call should be resumed or aborted. By returning the `MASTERCALL` constant from the PRECALL handler, for example, IP-MON instructs the master variant to resume the original call and the slave variants to abort the original call. Alternatively, the original call may be resumed or aborted in all variants.

Finally, IP-MON calls the POSTCALL handler. Here, the master variant copies its system call return values into the RB.

The slave variants instead wait for the return values to appear in the RB. Depending on the aforementioned system call metadata, the handler may wait using either a spin-wait loop, if the system call was not expected to block, or a specialized condition variable, whose implementation we describe in Section 8.5.1.7.

#### 8.5.1.4 System Call Monitoring Policies

There are many ways to draw the line between system calls to be monitored by GHUMVEE and system calls to be handled by IP-MON. We propose two concrete monitoring relaxation policies.

The first option is *spatial exemption*, where certain system calls are either unconditionally handled by IP-MON and not monitored by GHUMVEE, or handled by

IP-MON only if their system call arguments meet certain criteria. IP-MON comes with several predefined levels of spatial exemption, which the program developer or administrator can choose from. However, regardless of which level of spatial exemption is selected, GHUMVEE always monitors system calls that relate to allocation and management of process resources and threads, as we consider these system calls dangerous no matter what. These system calls include all signal-handling system calls as well as those that (i) allocate, manage, and close file descriptors (FDs); (ii) map, manage, and unmap memory regions; and (iii) create, control, and kill threads and processes. We refer to our earlier work for a full overview of IP-MON's spatial exemption policies [Volckaert et al. 2016].

The second option is *temporal exemption*, where IP-MON probabilistically exempts system calls from the monitoring policy if similar calls have been repeatedly approved by the monitor. We observe that many programs, especially those with high system call frequencies, often repeatedly invoke the same sequence of system calls. If a series of system calls is approved by GHUMVEE, then one possible temporal relaxation policy is to stochastically exempt some fraction of the identical system calls that follow within some time window or range. Note that temporal relaxation policies must be highly unpredictable; deterministic policies (e.g., "Exempt system calls X, Y, Z from monitoring after N approvals within an M millisecond time window") are insecure. In other words, care must be taken to ensure that temporal relaxation does not allow adversaries to coerce the MVEE into a state where potentially dangerous system calls are not monitored.

#### 8.5.1.5 IP-MON Initialization

IK-B does not forward any system calls to IP-MON until IP-MON explicitly registers itself through a new system call we added to the kernel. When this call is invoked, the kernel first attempts to report the call to GHUMVEE, which receives the notification and can decide if it wants to allow IP-MON to register.

The registration system call expects three arguments. The first argument is the set of "unmonitored" calls supported by IP-MON. If the IP-MON registration succeeds, IK-B forwards any system call in this set to IP-MON from that point onward, as we explained earlier. GHUMVEE can modify this set of system calls or, potentially, prevent the registration altogether. The second and third arguments are a pointer to the RB and a pointer to the entry point function that should be invoked when IK-B forwards a call to IP-MON.

The RB pointer must be valid and must point to a writable region. IP-MON must therefore set up an RB that it shares with all the other variants. We use the System V IPC facilities to create, initialize, and map the RB [man-pp. project 2017b].

GHUMVEE arbitrates the RB initialization process to ensure that all the variants attach to the same RB.

#### 8.5.1.6 The IP-MON File Map

GHUMVEE arbitrates all system calls that create/modify/destroy FDs, including sockets. It thus maintains metadata, such as the type of each FD (regular/pipe/socket/poll-fd/special). It also tracks which FDs are in non-blocking mode. System calls that operate on non-blocking FDs always return immediately, regardless of whether or not the corresponding operation succeeds.

Variants can map a read-only copy of this metadata into their address spaces using the same mechanism we use for the RB. We refer to this metadata as the IP-MON file map. We maintain exactly 1 byte of metadata per FD, resulting in a page-sized file map. For some system calls, IP-MON uses the file map to determine if the call is to be monitored or not, as per the monitoring policy.

#### 8.5.1.7 Blocking System Calls

Its file map permits IP-MON to predict whether an unmonitored call can block or not. IP-MON handles blocking calls efficiently. If the master variant knows that a call will block, it instructs the slaves to wait on an optimized and highly scalable IP-MON condition variable (as opposed to a slower spin-read loop) until the results become available. IP-MON uses the `futex(7)` API to implement wait and wake operations. This allowed us to implement several optimizations.

For each system call invocation, IP-MON allocates a separate structure within the RB. Each individual structure contains a condition variable. Slave variants must wait on only the condition variable associated with the system call results they are interested in. Using separate condition variables for each system call invocation prevents an unnecessary bottleneck that would arise when using just a single variable, because the slave variants might progress at different paces. Furthermore, IP-MON tracks whether or not there are variants waiting for the results of a specific system call invocation. If none are waiting when the master has finished writing its system call results into the buffer, no `FUTEX_WAKE` operation is needed to resume the slaves. IP-MON does not have to reuse condition variables because a new condition variable is allocated for each system call invocation. Thus, IP-MON does not have to reset condition variables to their initial state after it has used one to signal slave variants.

### 8.5.2 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 variants; (ii) enforcing lockstep, consistent, monitored execution of all system calls in all variants to prevent malicious impact of a single compromised variant on the monitor; and (iii) diversity among the variants to increase the likelihood that attacks cause observable divergence, i.e., that they fail to compromise the variants in consistent ways.

With those three properties in place, it becomes 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 through side-channel attacks. This is similar to many other code-reuse mitigations such as software diversity, software fault isolation [Wahbe et al. 1993], and control-flow integrity [Abadi et al. 2005a].

In ReMon, monitored system calls are still handled by a CP monitor, so malicious monitored calls are as hard to abuse as they are in existing CP MVEEs. For unmonitored calls, IP-MON relaxes the first two of the above three properties. The master variants can run ahead of the slaves and the system call consistency checks in the slaves' IP-MON, so an attacker could try to hijack the master's control with a malicious input to execute at least one, and possibly multiple, unmonitored calls without verification by a slave's IP-MON. An attacker could also attempt to locate the RB and feed malicious data to the slaves, in order to stall them or to tamper with their consistency checks. This way, the attacker could increase the window of opportunity to execute unmonitored calls in the master.

As long as the attacker executes unmonitored calls only according to a given relaxation policy, those capabilities by definition pose no significant security threat: unmonitored calls are exactly those calls that are defined by the chosen policy to pose either no security threat at all or an acceptable security risk. However, an attacker can 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. Therefore, we now consider several aspects of these attack scenarios.

**Unmonitored Execution of System Calls.** ReMon ensures that IP-MON can only execute unmonitored system calls if it is invoked by IK-B through its intended system call entry point. When invoked properly, IP-MON performs policy verification checks on conditionally allowed system calls, as well as the security checks a CP monitor normally performs. An attacker that manages to compromise a program variant could jump over these checks in an attempt to execute unmonitored system

calls directly. Such an attack would, however, be ineffective thanks to the authorization mechanism we described in Section 8.5.1.1.

**Manipulating the RB.** We designed IP-MON so that it never stores a pointer to the RB, nor any pointer derived thereof, in user-space-accessible memory. Instead, IK-B passes an RB pointer to IP-MON, and IP-MON keeps the RB pointer in a fixed register. To access the RB, the attacker must therefore find its location by random guessing or by mounting side-channel attacks.

ReMon's current implementation uses RBs that are 16 MiB and located on different addresses in each variant. This gives the RB pointer 24 bits of entropy per variant, which makes guessing attacks unlikely to succeed.

Furthermore, because neither IP-MON nor the application needs to know the exact location of the RB and because every invocation of IP-MON is routed through IK-B, we could extend IK-B to periodically move the RB to a different virtual address by modifying the variants' page table entries. This would further decrease the chances of a successful guessing attack.

**Diversified Variants.** Our current implementation of ReMon deploys the combined diversification of ASLR and DCL [Volckaert et al. 2015]. ReMon, however, support all other kinds of automated software diversity techniques as well. We refer to the literature for an overview of such techniques [Larsen et al. 2014]. The security evaluations in the literature, including demonstrations of resilience against concrete attacks, therefore still apply to ReMon.

## 8.6 Evaluation

We performed both performance and functional correctness evaluations of the different optimization techniques and protection schemes we implemented for GHUMVEE. We consider GHUMVEE as our baseline MVEE, against which we compare the spatial relaxation as implemented by ReMon.

### 8.6.1 Baseline GHUMVEE

To ensure that the approaches and design decisions taken in the development of GHUMVEE are applicable to a wide range of real-life programs, we have evaluated the functional correctness and the overhead imposed by our baseline MVEE on a wide range of applications. This includes not only the typical computationally heavy benchmarks, such as SPEC CPU2006, but also server applications and graphical applications. Unless otherwise mentioned, we evaluated our baseline MVEE with GHUMVEE monitoring two variants with DCL enabled.

All SPEC benchmarks can successfully run on top of GHUMVEE without the need to apply any patches. 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. When ASLR is enabled, the uninitialized data differs from one variant to another. In GHUMVEE, we whitelisted the responsible system call to prevent the false positive. The average overhead on SPEC CPU2006 is 7.2%.

We also 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 `MPlayer`. None of these programs needed patches to run on top of GHUMVEE.

The method of overriding the mapping type for file-backed shared memory was necessary to support KDE applications. 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. This overriding method will likely not work for all programs, however.

Our decision to disallow read/write shared mappings and the use of the System V IPC API in commodity applications does not constitute a big problem either. While shared memory is the preferred method for graphical applications to communicate with the display server, we have not seen a single application that does not have a fallback method in place for when GHUMVEE's monitor rejects the request to map shared memory pages. This fallback method typically yields significantly worse performance but is still acceptable in many situations. MPlayer, 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.

Early on in GHUMVEE's development, we also tested Firefox and LibreOffice. For LibreOffice, we tested operations such as opening and saving files, editing various types of documents, running the spell checker, etc. For Firefox, we tested opening several web pages. We repeated tests in which GHUMVEE spawned between one and four variants from the same executable, and tests were conducted with and without ASLR enabled. Although these tests were successful, both LibreOffice and Firefox needed extensive patching to be able to run in the context of an MVEE. These patches were needed to eliminate (benign) data races and address-sensitive behavior (cf. Section 8.3.5) and to embed our implicit-input replication agents and synchronization agents.

We not only verified the functional correctness but also evaluated the usability of interactive and real-time applications. Except for small start-up overheads, no significant usability impact was observed. For example, with two variants and

without hardware support, 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.

Finally, while we can apply our DCL protection with each variant using the same program binary, we did also successfully run programs for which we created a different diversified program binary for each variant. While we only created program binaries to which we applied code randomization techniques [Larsen et al. 2014], we believe that there are no fundamental limitations to the types of diversity techniques GHUMVEE can support.

### 8.6.2 Disjoint Code Layouts

We evaluated the DCL protection with GHUMVEE on two machines. The first machine has two Intel Xeon E5-2650L CPUs with 8 physical cores and a 20 MB 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 an 8 MB 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 hyperthreading and all dynamic frequency and voltage scaling features. Furthermore, we compiled both kernels with a 1,000 Hz tick rate to minimize the monitor-variant interaction latency.

#### 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 five times on their reference inputs. From each set of five measurements, we eliminated the first one to account for I/O-cache warm-up. On the 64-bit machine, we 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 two variants does not improve DCL's protection, we have also included the benchmark results for three and four variants for the sake of completeness.

As shown in Figures 8.5 and 8.6, the runtime overhead of DCL is rather low overall.[4] On our 32-bit machine, the average overhead across all SPEC benchmarks

4. The 434.zeusmp benchmark maps a very large code section and therefore does not run with more than two variants on our 32-bit machine.

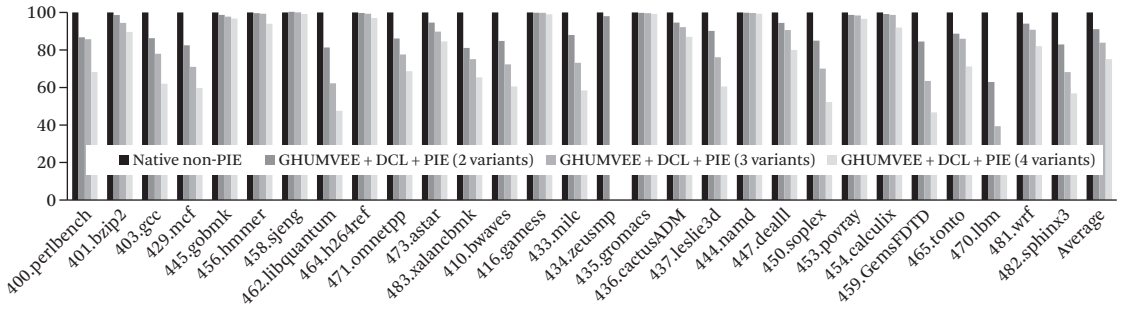


**Figure 8.5** Relative performance of 32-bit protected SPEC 2006 benchmarks.

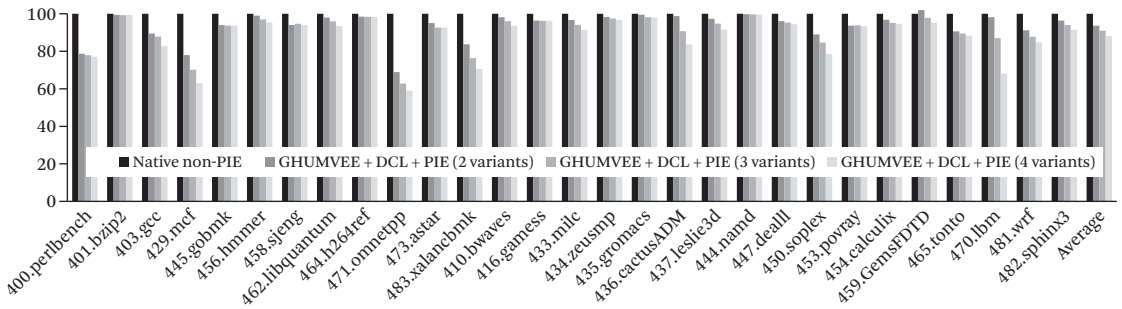


**Figure 8.6** Relative performance of 64-bit protected SPEC 2006 benchmarks.

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 [Jaleel 2007].

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%, respectively, by simply using PIE.

### 8.6.3 ReMon and IP-MON

We evaluated the performance of IP-MON's spatial relaxation policy on both synthetic benchmark suites and on a set of server benchmarks. We conducted all of our experiments on a machine with two 8-core Intel Xeon E5-2660 processors each having 20 MB of cache, 64 GB of RAM, and a gigabit Ethernet connection, running the x86_64 version of Ubuntu 14.04.3 LTS. This machine runs the Linux 3.13.11 kernel, to which we applied the IK-B patches. We used the official 2.19 versions of GNU's `glibc` and `libpthreads` in our experiments, but we did apply a small patch of less than 10 LoC to `glibc` to reinitialize IP-MON's thread-local storage variables after each fork. As before, we disabled hyper-threading as well as frequency and voltage scaling to maximize reproducibility of our measurements.

Address space layout randomization was enabled in our tests, and we configured ReMon to map IP-MON and its associated buffers at non-overlapping addresses in all variants.

#### Synthetic Benchmark Suites

We evaluated ReMon on the PARSEC 2.1, SPLASH-2x, and Phoronix benchmark suites[5]. These benchmarks cover a wide range in system call densities and patterns (e.g., bursty vs. spread over time, and mixes of sensitive and non-sensitive calls) as well as various scales and schemes of multi-threading, the most important factors contributing to the overhead of traditional CP-MVEEs that we want to overcome with IP-MON.

We evaluated all five levels of our spatial exemption policy on some of the Phoronix benchmarks, and show the performance of the `NONSOCKET_RW_LEVEL` policy on the other suites. We used the largest available input sets for all benchmarks and ran the multi-threaded benchmarks with four worker threads and used two variants for all benchmarks. We excluded PARSEC's `canneal` benchmark from our measurements because it purposely causes data races that result in divergent behavior when running multiple variants. This makes the benchmark incompatible with MVEEs. We also excluded SPLASH's `cholesky` benchmark due to incompatibilities with the version of the `gcc` compiler we used.

The results for these benchmarks are shown in Figures 8.7 and 8.8. The baseline overhead was measured by running ReMon with IP-MON and IK-B disabled. In this configuration, GHUMVEE runs as a stand-alone MVEE.

5. C. Segulja kindly provided his data race patches for PARSEC and SPLASH [Segulja and Abdelrahman 2014].

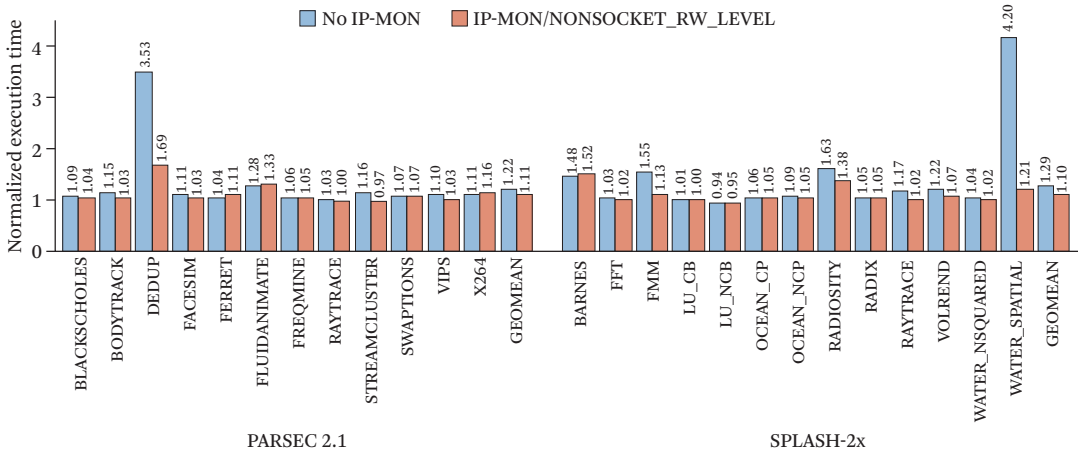


**Figure 8.7** Performance overhead for PARSEC 2.1 and SPLASH-2x benchmark suites (two variants).

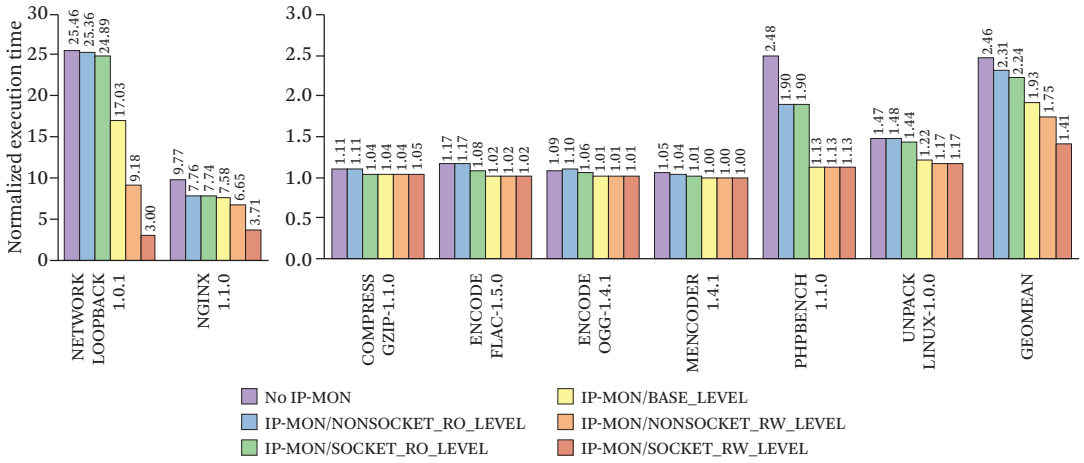


**Figure 8.8** Comparison of IP-MON's spatial relaxation policies in a set of Phoronix benchmarks (two variants).

GHUMVEE generally performs well in these benchmarks. Our machine can run the variants on disjoint CPU cores, which means that only the additional pressure on the memory subsystem and the MVEE itself cause performance degradation compared to the benchmarks' native performance. Yet, we still see the effect of enabling IP-MON. For PARSEC 2.1, the relative performance overhead decreases

from 21.9% to 11.2%. For SPLASH-2x, the overhead decreases from 29.2% to 10.4%. In Phoronix, the overhead drops from 146.4% to 41.2%. Particularly interesting are the `dedup` (PARSEC 2.1), `water_spatial` (SPLASH-2x) and `network_loopback` (Phoronix) benchmarks, which feature very high system call densities of over 60 K system call invocations per second. In these benchmarks, the overheads drop from 252.9% to 69.4%, from 320% to 20.7%, and from 2446% to 200%, respectively. Furthermore, the Phoronix results clearly show that different policies allow for different security-performance trade-offs.

### Server Benchmarks

Server applications are great candidates for execution and monitoring by MVEEs because they are frequently targeted by attackers and they often run on many-core machines with idle CPU cores that can run variants in parallel. In this section, we specifically evaluate our MVEE on applications used to evaluate other MVEEs. These applications include the `Apache` web server (used to evaluate Orchestra [Salamat et al. 2009]), `thttpd (ab)` and `lighttpd (ab)` (used to evaluate Tachyon [Maurer and Brumley 2012]), `lighttpd (http_load)` (used to evaluate Mx [Hosek and Cadar 2013]), and `beanstalkd`, `lighttpd (wrk)`, `memcached`, `nginx (wrk)`, and `redis` (used to evaluate VARAN [Hosek and Cadar 2015]). We use the same client and server configurations 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 0.125ms). 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 U.S. are 24–63 ms [Commission 2014]). 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 [man-pp. project 2017a].

Figure 8.9 shows the unlikely and the realistic scenarios side by side. For each benchmark, we measured the overhead IP-MON introduces when running between two and seven parallel variants with the spatial exemption policy at the `SOCKET_RW_LEVEL`. We also show the overhead for running two variants with IP-MON disabled. The latter case represents the best-case scenario without IP-MON.

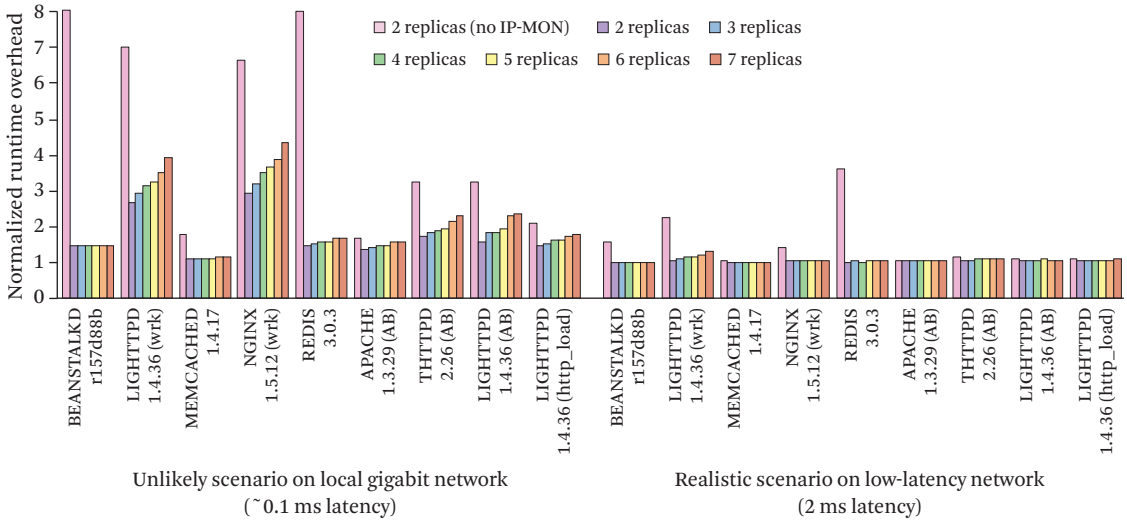


**Figure 8.9** Server benchmarks in two network scenarios for two to seven variants with IP-MON and two variants without IP-MON.

### 8.6.4 Comparison to Existing MVEEs

Table 8.2 compares GHUMVEE's and ReMon's performance with the results reported for other MVEEs in the literature [Hosek and Cadar 2013, Hosek and Cadar 2015, Maurer and Brumley 2012, Salamat et al. 2009]. We omitted some MVEEs from this comparison because we could not find (i) enough published performance results for these MVEEs or (ii) sufficient information about the setup in which these MVEEs were evaluated to allow for a meaningful comparison with ReMon and GHUMVEE.

Since each MVEE was evaluated in a different experimental setup, the table also lists two features that have a significant impact on the performance overhead. These are the network latencies, because higher latencies hide server-side overhead, and the CPU cache sizes, as some of the memory-intensive SPEC benchmarks benefit significantly from larger caches, in particular with multiple concurrent variants.

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 resided in the same rack and were hence connected by a very-low-latency gigabit Ethernet.

**Table 8.2** Comparison of Existing MVEEs (two variants)

| Orientation | Reliability MVEE | | | | | | Security MVEE | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Tachyon | | | Mx | | VARAN | Orchestra | GHUMVEE | ReMon | |
| Network | localhost | local few hops | coast-to-coast | localhost | USA-UK (150 ms) | same rack gigabit | local gigabit | local gigabit | local gigabit | local gigabit (5 ms) |
| CPU cache size | | | | . . . 8 MB . . . | | 8 MB | | 20 MB | . . . 20 MB . . . | |
| Reported overheads: | | | | | | | | | | |
| apache (ab) | | | | | | **2.4%** | 50% | 70% | 34% | **2.4%** |
| lighttpd (ab) | 790% | 272% | 30% | | | **0%** | | 226% | 55% | **0%** |
| thttpd (ab) | 1320% | 17% | **0%** | | | **0%** | | 223% | 73% | 2.7% |
| lighttpd (httpld) | | | | 249% | 4% | **1.0%** | | 108% | 45% | 3.5% |
| redis | | | | 1572% | 5% | 6% | | 1148% | 45% | **0.1%** |
| beanstalkd | | | | | | 52% | | 1259% | 45% | **0.6%** |
| memcached | | | | | | 14% | | 76% | **8.4%** | **0.3%** |
| nginx (wrk) | | | | | | 28% | | 564% | 194% | **0.8%** |
| lighttpd (wrk) | | | | | | 12% | | 603% | 169% | **0.7%** |
| SPEC CPU2006 | | | | 17.9% | | | | 7.2% | **3.1%** | |
| SPECint 2006 | | | | 17.6% | | 14.2% | | 12.1% | **3.9%** | |
| SPECfp 2006 | | | | 18.3% | | | | 3.8% | **2.5%** | |

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 unlikely worst-case scenarios for servers, 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 on the U.S. West Coast, while the other was located in England (Mx) or the U.S. 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 non-hardware-assisted security-oriented MVEEs while approaching the efficiency of reliability-oriented MVEEs.

## 8.7 Conclusion

In this chapter, we presented GHUMVEE, the most efficient non-hardware-assisted security-oriented MVEE to date. GHUMVEE is equipped to support a wide range of realistic programs, including those that contain user-space thread synchronization operations and address-sensitive data structures.

GHUMVEE supports disjoint code layouts, a practical technique to stop code-reuse attacks that rely on payloads containing absolute code addresses. It also supports relaxation policies and selective lockstepping, two techniques that further boost GHUMVEE's efficiency.

### Acknowledgments

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

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124, FA8750-15-C-0085, and FA8750-10-C-0237, by the National Science Foundation under award number CNS-1513837 as well as gifts from Mozilla, Oracle, and Qualcomm.

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

# References


M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. 2005a. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, pp. 340–353. DOI: 10.1145/1102120.1102165. 12, 25, 38, 39, 62, 82, 86, 95, 97, 110, 114, 117, 139, 141, 173, 174, 181, 186, 211, 233, 243, 249

M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. 2005b. A theory of secure control flow. In *Proceedings of the 7th International Conference on Formal Methods and Software Engineering (ICFEM)*. DOI: 10.1007/11576280_9. 182, 186

M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. 2009. Control-flow integrity: Principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1). DOI: 10.1145/1609956.1609960. 181, 189, 208

A. Acharya and M. Raje. 2000. MAPbox: Using parameterized behavior classes to confine untrusted applications. In *Proceedings of the 9th USENIX Security Symposium (SSYM)*, pp. 1–17. 16

P. Akritidis. 2010. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, pp. 177–192. 84, 173, 178

P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. 2008. Preventing memory error exploits with WIT. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P)*, pp. 263–277. DOI: 10.1109/SP.2008.30. 8, 58, 82, 84, 114, 173, 176, 178

P. Akritidis, M. Costa, M. Castro, and S. Hand. 2009. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, pp. 51–66. 84, 173, 178

Aleph One. 1996. Smashing the stack for fun and profit. *Phrack*, 7. 11, 17

A. Alexandrov, P. Kmiec, and K. Schauser. 1999. Consh: Confined execution environment for Internet computations. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.488. DOI: 10.1.1.57.488. 16

G. Altekar and I. Stoica. 2010. Focus replay debugging effort on the control plane. In *USENIX Workshop on Hot Topics in Dependability*. 89

S. Andersen and V. Abella. August 2004. Changes to functionality in Windows XP service pack 2—part 3: Memory protection technologies. http://technet.microsoft.com/en-us/library/bb457155.aspx. 9, 19, 184

J. Ansel. March 2014. Personal communication. 53

J. Ansel, P. Marchenko, Ú. Erlingsson, E. Taylor, B. Chen, D. Schuff, D. Sehr, C. Biffle, and B. Yee. 2011. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 355–366. DOI: 10.1145/1993316.1993540. 58, 59

O. Arias, L. Davi, M. Hanreich, Y. Jin, P. Koeberl, D. Paul, A.-R. Sadeghi, and D. Sullivan. 2015. HAFIX: Hardware-assisted flow integrity extension. In *Proceedings of the 52nd Design Automation Conference (DAC)*, pp. 74:1–74:6. DOI: 10.1145/2744769.2744847. 182, 208, 209

J.-P. Aumasson and D. J. Bernstein. 2012 SipHash: A fast short-input PRF. In *13th International Conference on Cryptology in India (INDOCRYPT)*. 73

M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. 2014. You can run but you can't read: Preventing disclosure exploits in executable code. In *ACM Conference on Computer and Communications Security (CCS)*. DOI: 10.1145/2660267.2660378, pp. 1342–1353. 65, 173, 177

M. Backes and S. Nürnberger. 2014. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *23rd USENIX Security Symposium*, pp. 433–447. 64, 66

A. Balasubramanian, M. S. Baranowski, A. Burtsev, and A. Panda. 2017. System programming in Rust: Beyond safety. In *Workshop on Hot Topics in Operating Systems (HotOS)*, pp. 94–99. DOI: 10.1145/3102980.3103006. 79

C. Basile, Z. Kalbarczyk, and R. Iyer. 2002. A preemptive deterministic scheduling algorithm for multithreaded replicas. In *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 149–158. DOI: 10.1109/DSN.2003.1209926. 230

C. Basile, Z. Kalbarczyk, and R. K. Iyer. 2006. Active replication of multithreaded applications. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 17(5):448–465. DOI: 10.1109/TPDS.2006.56. 230

A. Basu, J. Bobba, and M. D. Hill. 2011. Karma: Scalable deterministic record-replay. In *Proceedings of the International Conference on Supercomputing*, pp. 359–368. DOI: 10.1145/1995896.1995950. 230

M. Bauer. 2006. Paranoid penguin: an introduction to Novell AppArmor. *Linux J.*, (148):13. 16

A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. 2012. Dune: Safe user-level access to privileged CPU features. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 335–348. 213

T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. 2010. CoreDet: A compiler and runtime system for deterministic multithreaded execution. *ACM SIGARCH Computer Architecture News*, 38(1):53–64. DOI: 10.1145/1735971.1736029. 230

E. Berger, T. Yang, T. Liu, and G. Novark. 2009. Grace: Safe multithreaded programming for C/C++. *ACM Sigplan Notices*, 44(10):81–96. 230

E. D. Berger and B. G. Zorn. 2006. DieHard: Probabilistic memory safety for unsafe languages. *ACM SIGPLAN Notices*, (6):158–168. DOI: 10.1145/1133255.1134000. 214

E. Bhatkar, D. C. Duvarney, and R. Sekar. 2003. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the USENIX Security Symposium (SSYM)*, pp. 105–120. 10

S. Bhatkar and R. Sekar. 2008. Data space randomization. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pp. 1–22. DOI: 10.1007/978-3-540-70542-0_1. 11, 85

S. Bhatkar, R. Sekar, and D. C. DuVarney. 2005. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium (SSYM)*, pp. 17–17. http://dl.acm.org/citation.cfm?id=1251398.1251415. 10, 95

D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. 2015. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 268–279. DOI: 10.1145/2810103.2813691. 11

A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. 2014. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, pp. 227–242. DOI: 10.1109/SP.2014.22. 62, 140, 141, 182, 239

D. Blazakis. 2010. Interpreter exploitation. In *Proceedings of the 4th USENIX Conference on Offensive Technologies*, pp. 1–9. 59

T. Bletsch, X. Jiang, and V. Freeh. 2011. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pp. 353–362. DOI: 10.1145/2076732.2076783. 208, 209

T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. 2011. Jump-oriented programming: A new class of code-reuse attack. In *Procceedings of the 6th ACM Symposium on Information, Computer, and Communications Security (ASIACCS)*, pp. 30–40. DOI: 10.1145/1966913.1966919. 20, 81, 82, 117

E. Bosman and H. Bos. 2014. Framing signals—a return to portable shellcode. In *IEEE Symposium on Security and Privacy (S&P)*, pp. 243–258. DOI: 10.1109/SP.2014.23. 31, 140

K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi. 2016. Leakage-resilient layout randomization for mobile devices. In *23rd Annual Network and Distributed System Security Symposium (NDSS)*. 68

S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina. 2011. Exploit programming: From buffer overflows to "weird machines" and theory of computation. Usenix ;login: issue: December 2011, volume 36, number 6. 19

E. Buchanan, R. Roemer, H. Shacham, and S. Savage. 2008. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, pp. 27–38. DOI: 10.1145/1455770.1455776. 233

M. Budiu, Ú. Erlingsson, and M. Abadi. Architectural support for software-based protection. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, pp. 42–51. DOI: 10.1145/1181309.1181316. 208, 209

N. Burow, S. A. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz. 2016. Control-flow integrity: Precision, security, and performance. *Computing Research Repository (CoRR)*. 50(1). http://arxiv.org/abs/1602.04056. 12, 28, 62, 82

N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. 2017. Control-flow integrity: precision, security, and performance. *ACM Computing Surveys*. DOI: 10.1145/3054924. 12

J. Butler and anonymous. 2004. Bypassing 3rd party Windows buffer overflow protection. *Phrack*, 11. 17

N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. 2015. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Security Symposium*, pp. 161–176. http://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini. 15, 20, 21, 59, 81, 82, 97, 137, 182, 183, 185, 186, 188, 200, 204, 211

N. Carlini and D. Wagner. 2014. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium*, pp. 385–399. http://dl.acm.org/citation.cfm?id=2671225.2671250. 15, 53, 82, 84, 86, 97, 114, 137, 139, 140, 176, 177, 179, 182, 183, 184, 186, 188, 200, 202, 209

M. Castro, M. Costa, and T. Harris. 2006. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 147–160. 11, 86

M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. 2009. Fast byte-granularity software fault isolation. In *ACM Symposium on Operating Systems Principles*, pp. 45–58. DOI: 10.1145/1629575.1629581. 86, 93

L. Cavallaro. 2007. Comprehensive memory error protection via diversity and taint-tracking. PhD thesis, Universita Degli Studi Di Milano. 214, 223, 237

S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. 2010. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pp. 559–572. DOI: 10.1145/1866307.1866370. 20, 81, 117, 183, 184, 185, 186, 200, 202

S. Checkoway and H. Shacham. 2010. Escape from return-oriented programming: Return-oriented programming without returns (on the x86). Technical report CS2010-0954, UC San Diego. http://cseweb.ucsd.edu/~hovav/dist/noret.pdf. 200, 202

P. Chen, Y. Fang, B. Mao, and L. Xie. 2011. JITDefender: A defense against JIT spraying attacks. In *26th IFIP International Information Security Conference*, volume 354, pp. 142–153. 60

P. Chen, R. Wu, and B. Mao. 2013. JITSafe: A framework against just-in-time spraying attacks. *IET Information Security*, 7(4):283–292. DOI: 10.1049/iet-ifs.2012.0142. 59, 60

S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. 2005 Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium*. http://dl.acm.org/citation.cfm?id=1251398.1251410. 21, 184

X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. 2015. StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *Symposium on Network and Distributed System Security (NDSS)*. 173, 178

Y. Chen, D. Zhang, R. Wang, R. Qiao, A. M. Azab, L. Lu, H. Vijayakumar, and W. Shen. 2017. NORAX: Enabling execute-only memory for COTS binaries on AArch64. In *IEEE Symposium on Security and Privacy (S&P)*, pp. 304–319. DOI: 10.1109/SP.2017.30. 68

Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. 2014. ROPecker: A generic and practical approach for defending against ROP attacks. In *Proceedings of the 21st Synposium on Network and Distributed System Security (NDSS)*. 117, 118, 119, 127, 173, 176, 182, 209

M. Co, J. W. Davidson, J. D. Hiser, J. C. Knight, A. Nguyen-Tuong, W. Weimer, J. Burket, G. L. Frazier, T. M. Frazier, and B. Dutertre, et al. 2016. Double Helix and RAVEN: A system for cyber fault tolerance and recovery. In *Proceedings of the 11th Annual Cyber and Information Security Research Conference*, p. 17. DOI: 10.1145/2897795.2897805. 214

F. B. Cohen. 1993. Operating system protection through program evolution. *Computers & Security*, 12(6): 565–584. DOI: 10.1016/0167-4048(93)90054-9. 62

Corelan. 2011. Mona: A debugger plugin/exploit development Swiss army knife. http://redmine.corelan.be/projects/mona. 136

C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. 2000. SubDomain: Parsimonious server security. In *Proccedings of the 14th USENIX Conference on System Administration*, pp. 355–368. 16

C. Cowan, S. Beattie, J. Johansen, and P. Wagle. 2003. Pointguard™: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium (SSYM)*, pp. 7–7. http://dl.acm.org/citation.cfm?id=1251353.1251360. 11, 63, 76, 82, 85

C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, volume 81, pp. 346–355. 61, 63, 82, 95, 211, 233

B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. 2006. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th USENIX Security Symposium*, 9. 211, 213, 214, 217, 237

S. Crane, A. Homescu, and P. Larsen. 2016. Code randomization: Haven't we solved this problem yet? In *IEEE Cybersecurity Development (SecDev)*. DOI: 10.1109/SecDev.2016.036. 66

S. Crane, P. Larsen, S. Brunthaler, and M. Franz. 2013. Booby trapping software. In *New Security Paradigms Workshop (NSPW)*, pp. 95–106. DOI: 10.1145/2535813.2535824. 68

S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. 2015. Readactor: Practical code randomization resilient to memory disclosure. In *36th IEEE Symposium on Security and Privacy (S&P)*, pp. 763–780. DOI: 10.1109/SP.2015.52. 11, 60, 66, 76, 173, 178

S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz. 2015. It's a TRaP: Table randomization and protection against function-reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 243–255. DOI: 10.1145/2810103.2813682. 11, 68, 77, 159, 171, 173, 178

J. Criswell, N. Dautenhahn, and V. Adve. 2014. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *IEEE Symposium on Security and Privacy (S&P)*, pp. 292–307. DOI: 10.1109/SP.2014.26. 58

H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. 2013. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 388–405. DOI: 10.1145/2517349.2522735. 230

D. Dai Zovi. 2010. Practical return-oriented programming. Talk at *SOURCE Boston*, 2010. 117

L. Dalessandro, D. Dice, M. Scott, N. Shavit, and M. Spear. 2010. Transactional mutex locks. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part II*, pp. 2–13. 44

T. H. Y. Dang, P. Maniatis, and D. Wagner. 2015. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer, and Communications Security (ASIACCS)*, pp. 555–566. DOI: 10.1145/2714576.2714635. 10, 137, 208

DarkReading. November 2009. Heap spraying: Attackers' latest weapon of choice. http://www.darkreading.com/security/vulnerabilities/showArticle.jhtml?articleID=221901428. 133

L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nurnberger, and A.-R. Sadeghi. 2012. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*. 58, 208

L. Davi, P. Koeberl, and A.-R. Sadeghi. 2014. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Annual Design Automation Conference—Special Session: Trusted Mobile Embedded Computing (DAC)*, pp. 1–6. DOI: 10.1145/2593069.2596656. 173, 174, 209

L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. 2014. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium*, pp. 401–416. http://dl.acm.org/citation.cfm?id=2671225.2671251. 15, 43, 53, 82, 84, 86, 97, 114, 139, 140, 169, 174, 176, 177, 179, 182, 183, 184, 186, 188, 200, 209, 211

L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. 2015. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *22nd Annual Network and Distributed System Security Symposium (NDSS)*. DOI: 10.14722/ndss.2015.23262. 64

L. Davi, A.-R. Sadeghi, and M. Winandy. 2011. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *ACM Symposium on Information, Computer, and Communications Security (ASIACCS)*, pp. 40–51. DOI: 10.1145/1966913.1966920. 139, 141

L. de Moura and N. Bjørner. 2009. Generalized, efficient array decision procedures. In *Formal Methods in Computer Aided Design (FMCAD)*. DOI: 10.1109/FMCAD.2009.5351142. 161

L. M. de Moura and N. Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 337–340. 140, 161

T. de Raadt. 2005. Exploit mitigation techniques. http://www.openbsd.org/papers/ven05-deraadt/index.html. 8

J. Dean, D. Grove, and C. Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming (ECOOP)*, pp. 77–101. 32

D. Dechev. 2011. The ABA problem in multicore data structures with collaborating operations. In *7th International Conference on Collaborative Computing: Networking, Applications, and Worksharing (CollaborateCom)*, pp. 158–167. DOI: 10.4108/icst.collaboratecom.2011.247161. 44

L. Deng, Q. Zeng, and Y. Liu. 2015. ISboxing: An instruction substitution based data sandboxing for x86 untrusted libraries. In *30th International Conference on ICT Systems Security and Privacy Protection*, pp. 386–400. 41

L. P. Deutsch and A. M. Schiffman. 1984. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 297–302. DOI: 10.1145/800017.800542. 54

J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. 2008. HardBound: Architectural support for spatial safety of the C programming language. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 103–114. DOI: 10.1145/1353534.1346295. 109

J. Devietti, B. Lucia, L. Ceze, and M. Oskin. 2009. DMP: Deterministic shared memory multiprocessing. *ACM SIGARCH Computer Architecture News*, 37(1):85–96. DOI: 10.1145/1508244.1508255. 230

D. Dewey and J. T. Giffin. 2012. Static detection of C++ vtable escape vulnerabilities in binary code. In *Symposium on Network and Distributed System Security (NDSS)*. 171

D. Dhurjati, S. Kowshik, and V. Adve. June 2006. SAFECode: Enforcing alias analysis for weakly typed languages. *SIGPLAN Notices*, 41 (6): 144–157. DOI: 10.1145/1133255.1133999. 82, 84

U. Drepper. April 2006. SELinux memory protection tests. http://www.akkadia.org/drepper/selinux-mem.html. 238

V. D'Silva, M. Payer, and D. Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *LangSec'15: Second Workshop on Language-Theoretic Security*. DOI: 10.1109/SPW.2015.33. 16

T. Durden. 2002. Bypassing PaX ASLR protection. *Phrack*, 11. 10, 17

EEMBC. The embedded microprocessor benchmark consortium: EEMBC benchmark suite. http://www.eembc.org. 206

Ú Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula. 2006. XFI: Software guards for system address spaces. In *Proceedings of the 7th USENIX Symposium on Operating System Design and Implementation*, pp. 75–88. 58, 86, 95

H. Etoh and K. Yoda. June 2000. Protecting from stack-smashing attacks. Technical report, IBM Research Division, Tokyo Research Laboratory. 63

C. Evans. 2013. Exploiting 64-bit Linux like a boss. http://scarybeastsecurity.blogspot.com/2013/02/exploiting-64-bit-linux-like-boss.html. 117

I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. E. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. 2015. Missing the point(er): On the effectiveness of code pointer integrity. In *36th IEEE Symposium on Security and Privacy, (S&P)*, pp. 781–796. DOI: 10.1109/SP.2015.53. 11, 62, 87

I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. 2015. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 901–913. DOI: 10.1145/2810103.2813646. 20, 21, 59, 82, 97, 137, 211

Federal Communications Commission. 2014. Measuring broadband America—2014. http://www.fcc.gov/reports/measuring-broadband-america-2014. 256

C. Fetzer and M. Suesskraut. 2008. SwitchBlade: Enforcing dynamic personalized system call models. In *Proceedings of the 3rd European Conference on Computer Systems*, pp. 273–286. DOI: 10.1145/1357010.1352621. 16

A. Fokin, E. Derevenetc, A. Chernov, and K. Troshina. 2011. SmartDec: Approaching C++ decompilation. In *Working Conference on Reverse Engineering (WCRE)*. 171

B. Ford and R. Cox. 2008. Vx32: Lightweight user-level sandboxing on the x86. In *Proceedings of the USENIX ATC*, pp. 293–306. 8, 9

M. Frantzen and M. Shuey. 2001. StackGhost: Hardware facilitated stack protection. In *USENIX Security Symposium*. 139, 141

I. Fratric. 2012. Runtime prevention of return-oriented programming attacks. http://github.com/ivanfratric/ropguard/blob/master/doc/ropguard.pdf. 139, 173, 176

Gaisler Research. LEON3 synthesizable processor. http://www.gaisler.com. 183, 206

A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 465–478. DOI: 10.1145/1543135.1542528. 50

T. Garfinkel, B. Pfaff, and M. Rosenblum. 2004. Ostia: A delegating architecture for secure system call interposition. In *Network and Distributed System Security Symposium (NDSS)*. 241

R. Gawlik and T. Holz. 2014. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Annual Computer Security Applications Conference (ACSAC)*, pp. 396–405. 171, 173, 176, 182

R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz. 2016. Enabling client-side crash-resistance to overcome diversification and information hiding. In *23rd Annual Network and Distributed System Security Symposium (NDSS)*. 68

X. Ge, M. Payer, and T. Jaeger. 2017. An evil copy: How the loader betrays you. In *Network and Distributed System Security Symposium (NDSS)*. DOI: 10.14722/ndss.2017.23199 . 15

J. Gionta, W. Enck, and P. Ning. 2015. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *5th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pp. 325–336. DOI: 10.1145/2699026.2699107. 65

GNU.org. The GNU C library: Environment access. http://www.gnu.org/software/libc/manual/html_node/Environment-Access.html. 220

E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. 2014a. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, pp. 575–589. DOI: 10.1109/SP.2014.43. 15, 53, 82, 84, 86, 97, 114, 124, 125, 126, 129, 134, 136, 137, 139, 140, 174, 175, 177, 182, 183, 186, 188, 200, 202, 211

E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. 2014b. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the 23rd USENIX Security Symposium*. http://dl.acm.org/citation.cfm?id=2671225.2671252. 122, 139, 140, 169, 177, 179, 182, 186, 188, 209

E. Göktaş, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos. 2016. Undermining information hiding (and what to do about it). In *Proceedings of the 25th USENIX Security Symposium*, pp. 105–119. 11, 68

I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. 1996. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 6th USENIX Security Symposium (SSYM)*. 16

Google Chromium Project. 2013. Undefined behavior sanitizer. http://www.chromium.org/developers/testing/undefinedbehaviorsanitizer. 7

B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. 2017. ASLR on the line: Practical cache attacks on the MMU. In *Annual Network and Distributed System Security Symposium (NDSS)*. 67

Y. Guillot and A. Gazet. 2010. Automatic binary deobfuscation. *J. Comput. Virol.* 6(3): pp. 261–276. DOI: 10.1007/s11416-009-0126-4. 160

I. Haller, E. Göktaş, E. Athanasopoulos, G. Portokalidis, and H. Bos. 2015. ShrinkWrap: VTable protection without loose ends. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pp. 341–350. DOI: 10.1145/2818000.2818025. 136

I. Haller, Y. Jeon, H. Peng, M. Payer, H. Bos, C. Giuffrida, and E. van der Kouwe. 2016. TypeSanitizer: Practical type confusion detection. In *ACM Conference on Computer and Communication Security (CCS)*. DOI: 10.1145/2976749.2978405. 7

N. Hasabnis, A. Misra, and R. Sekar. 2012. Light-weight bounds checking. In *IEEE/ACM Symposium on Code Generation and Optimization*. DOI: 10.1145/2259016.2259034. 84

Hex-Rays. 2017. IDA Pro. http://www.hex-rays.com/index.shtml. 128

M. Hicks. 2014. What is memory safety? http://www.pl-enthusiast.net/2014/07/21/memory-safety/. 4

E. Hiroaki and Y. Kunikazu. 2001. ProPolice: Improved stack-smashing attack detection. *IPSJ SIG Notes*, pp. 181–188. 11

J. Hiser, A. Nguyen, M. Co, M. Hall, and J. W. Davidson. 2012. ILR: Where'd my gadgets go? In *33rd IEEE Symposium on Security and Privacy (S&P)*, pp. 571–585. DOI: 10.1109/SP.2012.39. 11, 66

J. D. Hiser, D. Williams, A. Filipi, J. W. Davidson, and B. R. Childers. 2006. Evaluating fragment construction policies for SDT systems. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE)*, pp. 122–132. DOI: 10.1145/1134760.1134778. 8, 9

U. Hölzle, C. Chambers, and D. Ungar. 1991. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *European Conference on Object-Oriented Programming (ECOOP)*, pp. 21–38. 54

U. Hölzle, C. Chambers, and D. Ungar. 1992. Debugging optimized code with dynamic deoptimization. In *Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 32–43. DOI: 10.1145/143103.143114. 54

A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. 2013. Librando: Transparent code randomization for just-in-time compilers. CCS '13, pp. 993–1004. DOI: 10.1145/2508859.2516675. 58, 59

A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. 2013. Profile-guided automated software diversity. In *IEEE/ACM Symposium on Code Generation and Optimization*, pp. 1–11. DOI: 10.1109/CGO.2013.6494997. 85

P. Hosek and C. Cadar. 2013. Safe software updates via multi-version execution. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE'13)*, pp. 612–621. DOI: 10.1109/ICSE.2013.6606607. 214, 256, 257

Petr Hosek and Cristian Cadar. 2015. 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)*, pp. 339–353. DOI: 10.1145/2694344.2694390. 214, 215, 218, 224, 227, 256, 257

H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. 2015. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium*, pp. 177–192. http://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/hu. 21

R. Hund, C. Willems, and T. Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symposium on Security and Privacy (S&P)*, pp. 191–205. DOI: 10.1109/SP.2013.23. 82, 86, 141

G. Hunt and D. Brubacher. 1999. Detours: Binary interception of win32 functions. In *Usenix Windows NT Symposium*, pp. 135–143. 232

Intel. 2013. *Intel Architecture Instruction Set Extensions Programming Reference*. http://download-software.intel.com/sites/default/files/319433-015.pdf. 108

Intel.2013. Introduction to Intel memory protection extensions. http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions. 93

Intel. 2013. *Intel 64 and IA-32 Architectures Software Developer's Manual—Combined Volumes 1, 2a, 2b, 2c, 3a, 3b, and 3c*. 178

Intel. 2014. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2: Instruction Set Reference, A–Z*. 2014. 223, 224

Itanium C++ ABI. http://mentorembedded.github.io/cxx-abi/abi.html. 32

A. Jaleel. 2007. Memory characterization of workloads using instrumentation-driven simulation—a pin-based memory characterization of the SPEC CPU2000 and SPEC CPU2006 benchmark suites. *technical report*. http://www.glue.umd.edu/~ajaleel/workload/. 253

D. Jang, Z. Tatlock, and S. Lerner. 2014. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Symposium on Network and Distributed System Security (NDSS)*. 32, 173, 176

jduck. 2010. The latest Adobe exploit and session upgrading. http://bugix-security.blogspot.de/2010/03/adobe-pdf-libtiff-working-exploitcve.html. 182

T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. 2002. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*. 5, 82, 84, 88, 95

N. Joly. 2013. Advanced exploitation of Internet Explorer 10/Windows 8 overflow (Pwn2Own 2013). http://www.vupen.com/blog/20130522.Advanced_Exploitation_of_IE10_Windows8_Pwn2Own_2013.php. 117, 124, 162

M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev. 2012. Branch regulation: Low-overhead protection from code reuse attacks. In *Proceedings of the 39th Annual*

*International Symposium on Computer Architecture (ISCA)*. http://dl.acm.org/citation.cfm?id=2337159.2337171. DOI: 10.1109/ISCA.2012.6237009. 182, 209

M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh. 2013. Scrap: Architecture for signature-based protection from code reuse attacks. In *IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013)*, pp. 258–269. DOI: 10.1109/HPCA.2013.6522324. 209

C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. 2006. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, pp. 339–348. DOI: 10.1109/ACSAC.2006.9. 11, 85

V. Kiriansky, D. Bruening, and S. P. Amarasinghe. 2002. Secure execution via program shepherding. In *Proceedings 11th USENIX Security Symposium*, pp. 191–206. 8, 9

K. Koning, H. Bos, and C. Giuffrida. 2016. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 431–442. DOI: 10.1109/DSN.2016.46. 211, 214, 217

T. Kornau. 2010. Return oriented programming for the ARM architecture. Ph.D. thesis, Master's thesis, Ruhr-Universitat Bochum. 233

V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. 2014a. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 147–163. 9, 10, 59, 62, 105, 106, 107, 173, 178, 179

V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. 2014b. Code-Pointer Integrity website. http://dslab.epfl.ch/proj/cpi/. 179

V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, and D. Song. 2015. Poster: Getting the point (er): On the feasibility of attacks on code-pointer integrity. In *36th IEEE Symposium on Security and Privacy (S&P)*. 87

P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. 2014. SoK: Automated software diversity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, pp. 276–291. DOI: 10.1109/SP.2014.25. 11, 62, 66, 250, 252

C. Lattner and V. Adve. 2005. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *ACM Conference on Programming Language Design and Implementation*, pp. 129–142. DOI: 10.1145/1064978.1065027. 91, 108

C. Lattner, A. Lenharth, and V. Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *ACM Conference on Programming Language Design and Implementation*, pp. 278–289. DOI: 10.1145/1273442.1250766. 91, 108

B. Lee, C. Song, T. Kim, and W. Lee. 2015. Type casting verification: Stopping an emerging attack vector. In *USENIX Security 15*, pp. 81–96. http://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lee. 7

D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. 2010. Respec: Efficient online multiprocessor replayvia speculation and external determinism. *ACM SIGARCH Computer Architecture News*, 38(1):77–90. DOI: 10.1145/1736020.1736031. 230

J. Lettner, B. Kollenda, A. Homescu, P. Larsen, F. Schuster, L. Davi, A.-R. Sadeghi, T. Holz, and M. Franz. 2016. Subversive-C: Abusing and protecting dynamic message dispatch. In *USENIX Annual Technical Conference (ATC)*, pp. 209–221. 70, 140

E. Levy. 1996. Smashing the stack for fun and profit. *Phrack*, 7. 61

J. Li, Z. Wang, T. K. Bletsch, D. Srinivasan, M. C. Grace, and X. Jiang. 2011. Comprehensive and efficient protection of kernel control data. *IEEE Transactions on Information Forensics and Security*, 6(4):1404–1417. DOI: 10.1109/TIFS.2011.2159712. 82, 86

C. Liebchen, M. Negro, P. Larsen, L. Davi, A.-R. Sadeghi, S. Crane, M. Qunaibit, M. Franz, and M. Conti. 2015. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM Conference on Computer and Communications Security (CCS)*. DOI: 10.1145/2810103.2813671. 182, 183, 205

Linux Man-Pages Project. 2017a. tc-netem(8)—Linux manual page. 256

Linux Man-Pages Project. 2017b. shmop(2)—Linux manual page. 247

*Linux Programmer's Manual*. 2017a. vdso(7)—Linux manual page. 223

*Linux Programmer's Manual*. 2017b. getauxval(3)—Linux manual page. 224

*Linux Programmer's Manual*. 2017c. signal(7)—Linux manual page. 225

T. Liu, C. Curtsinger, and E. Berger. 2011. DTHREADS: Efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating System Principles (SOSP)*, pp. 327–336. DOI: 10.1145/2043556.2043587. 230

LLVM. The LLVM compiler infrastructure. http://llvm.org/. 102

K. Lu, X. Zhou, T. Bergan, and X. Wang. 2014. Efficient deterministic multithreading without global barriers. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 287–300. DOI: 10.1145/2555243.2555252. 230

K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. 2015. ASLR-Guard: Stopping address space leakage for code reuse attacks. In *ACM Conference on Computer and Communications Security (CCS)*, pp. 280–291. DOI: 10.1145/2810103.2813694. 68

J. Maebe, M. Ronsse, and K. D. Bosschere. 2003. Instrumenting JVMs at the machine code level. In *3rd PA3CT symposium*, volume 19, pp. 105–107. 222

G. Maisuradze, M. Backes, and C. Rossow. 2003. What cannot be read, cannot be leveraged? Revisiting assumptions of JIT-ROP defenses. In *USENIX Security Symposium*. 67

M. Marschalek. 2014. Dig deeper into the IE vulnerability (cve-2014-1776) exploit. http://www.cyphort.com/dig-deeper-ie-vulnerability-cve-2014-1776-exploit/. 182

A. J. Mashtizadeh, A. Bittau, D. Mazieres, and D. Boneh. 2014. Cryptographically enforced control flow integrity. http://arxiv.org/abs/1408.1451. 86

A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. 2015. CCFI: Cryptographically enforced control flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, pp. 941–951. DOI: 10.1145/2810103.2813676. 72, 76, 77

M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell. 2013. System V application binary interface: AMD64 architecture processor supplement. http://x86-64.org/documentation/abi.pdf. 150

M. Maurer and D. Brumley. 2012. Tachyon: Tandem execution for efficient live patch testing. In *USENIX Security Symposium*, pp. 617–630. 214, 256, 257

S. McCamant and G. Morrisett. 2006. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th USENIX Security Symposium*. 41, 59, 68, 86

H. Meer. 2010. Memory corruption attacks: The (almost) complete history. In *Proceedings of Blackhat USA*. 62

T. Merrifield and J. Eriksson. 2013. Conversion: Multi-version concurrency control for main memory segments. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, pp. 127–139. DOI: 10.1145/2465351.2465365. 230

Microsoft Corp. November 2014. Enhanced mitigation experience toolkit (EMET) 5.1. http://technet.microsoft.com/en-us/security/jj653751. 173, 176

Microsoft Developer Network. 2017. Argument passing and naming conventions. http://msdn.microsoft.com/en-us/library/984x0h58.aspx. 149, 151, 154

V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. 2015. Opaque control-flow integrity. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*. http://www.internetsociety.org/doc/opaque-control-flow-integrity. 173, 177, 182

J. R. Moser. 2006. Virtual machines and memory protections. http://lwn.net/Articles/210272/. 238

G. Murphy. 2012. Position independent executables—adoption recommendations for packages. http://people.redhat.com/~gmurphy/files/pie.odt. 238

S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. 2012. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *International Symposium on Computer Architecture*, pp. 189–200. DOI: 10.1145/2366231.2337181. 109

S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. 2015. Everything you want to know about pointer-based checking. In *First Summit on Advances in Programming Languages (SNAPL)*. DOI: 10.4230/LIPIcs.SNAPL.2015.190. 5

S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM Sigplan Notices*, volume 44, pp. 245–258. DOI: 10.1145/1542476.1542504. 4, 5, 36, 82, 84, 88, 91, 97, 99, 101, 102, 110, 112, 211

S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. 2010. CETS: Compiler enforced temporal safety for C. In *ACM Sigplan Notices*, volume 45, pp. 31–40. DOI: 10.1145/1806651.1806657. 6, 83, 84, 88, 89, 91, 98, 108, 173, 178, 211

G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. 2005. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526. DOI: 10.1145/1065887.1065892. 5, 82, 84, 88, 95

Nergal. December 2001. The advanced return-into-lib(c) exploits (PaX case study). *Phrack*, 58 (4): 54. http://www.phrack.org/archives/58/p58_0x04_Advanced%20return-into-lib(c)%20exploits%20(PaX%20case%20study)_by_nergal.txt. 81, 82, 185, 203

B. Niu. 2015. Practical control-flow integrity. Ph.D. thesis, Lehigh University. 26, 37, 39, 56, 60

B. Niu and G. Tan. 2013. Monitor integrity protection with space efficiency and separate compilation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 199–210. DOI: 10.1145/2508859.2516649. 39, 82, 86, 173, 175

B. Niu and G. Tan. 2014a. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. DOI: 10.1145/2594291.2594295. 26, 27, 40, 44, 58, 82, 110, 114, 182

B. Niu and G. Tan. 2014b. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *ACM Conference on Computer and Communication Security (CCS)*, pp. 1317–1328. DOI: 10.1145/2660267.2660281. 9, 26, 34

B. Niu and G. Tan. 2015. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 914–926. DOI: 10.1145/2810103.2813644. 14, 30, 59

A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida. 2016. Poking holes in information hiding. In *25th USENIX Security Symposium*, pp. 121–138. 11, 68, 94

M. Olszewski, J. Ansel, and S. Amarasinghe. 2009. Kendo: Efficient deterministic multithreading in software. *ACM Sigplan Notices*, 44(3):97–108. DOI: 10.1145/1508244.1508256. 230

K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. 2010. G-Free: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, pp. 49–58. DOI: 10.1145/1920261.1920269. 173, 177, 210

V. Pappas, M. Polychronakis, and A. D. Keromytis. 2013. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Security Symposium*, pp. 447–462. 117, 118, 119, 127, 173, 176, 182, 209

A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. 2017. Marx: Uncovering class hiearchies in C++ programs. In *Annual Network and Distributed System Security Symposium (NDSS)*. 67

PaX Team. 2004a. Address space layout randomization. http://pax.grsecurity.net/docs/aslr.txt, 2004a. 82, 85, 211

PaX Team. 2004b PaX non-executable pp. design & implementation. http://pax.grsecurity.net/docs/noexec.txt, 2004b. 8, 211

M. Payer. 2012. Safe loading and efficient runtime confinement: A foundation for secure execution. Ph.D. thesis, ETH Zurich. http://nebelwelt.net/publications/12PhD. DOI: 10.1109/SP.2012.11. 8

M. Payer, A. Barresi, and T. R. Gross. 2015. Fine-grained control-flow integrity through binary hardening. In *Proceedings of the 12th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. DOI: 10.1007/978-3-319-20550-2_8. 10, 14, 58, 173, 174

M. Payer and T. R. Gross. 2011. Fine-grained user-space security through virtualization. In *Proceedings of the 7th International Conference on Virtual Execution Environments (VEE)*. DOI: 10.1145/1952682.1952703. 8, 9

A. Pelletier. 2012. Advanced exploitation of Internet Explorer heap overflow (Pwn2Own 2012 exploit). VUPEN Vulnerability Research Team (VRT) blog. http://www.vupen.com/blog/20120710.Advanced_Exploitation_of_Internet_Explorer_HeapOv_CVE-2012-1876.php. 124, 131

J. Pewny and T. Holz. 2013. Control-flow restrictor: Compiler-based CFI for iOS. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pp. 309–318. DOI: 10.1145/2523649.2523674. 58

Phoronix. Phoronix test suite. http://www.phoronix-test-suite.com/. 114

A. Prakash, X. Hu, and H. Yin. 2015. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Symposium on Network and Distributed System Security (NDSS)*. 58, 160, 170, 171, 173, 176, 182

N. Provos. 2003. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium SSYM)*, volume 12, pp. 18–18. http://dl.acm.org/citation.cfm?id=1251353.1251371. 16, 241

H. P. Reiser, J. Domaschka, F. J. Hauck, R. Kapitza, and W. Schröder-Preikschat. 2006. Consistent replication of multithreaded distributed objects. In *IEEE Symposium on Reliable Distributed Systems*, pp. 257–266. DOI: 10.1109/SRDS.2006.14. 230

R. Roemer, E. Buchanan, H. Shacham, and S. Savage. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34. DOI: 10.1145/2133375.2133377. 20, 117, 181, 185

R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, A.-R. Sadeghi, and H. Okhravi. 2017. Address oblivious code reuse: On the effectiveness of leakage resilient diversity. In *Annual Network and Distributed System Security Symposium (NDSS)*. 69

J. M. Rushby. 1981. Design and verification of secure systems. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 12–21. DOI: 10.1145/800216.806586. 214

M. Russinovich, D. A. Solomon, and A. Ionescu. 2012. *Windows Internals, Part 1*. Microsoft Press, 6th edition. ISBN 978-0-7356-4873-9. 155, 175

SafeStack. Clang documentation: Safestack. http://clang.llvm.org/docs/SafeStack.html. 102

B. Salamat. 2009. Multi-variant execution: Run-time defense against malicious code injection attacks. Ph.D. thesis, University of California at Irvine. 218

B. Salamat, T. Jackson, A. Gal, and M. Franz. 2009. 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)*, pp. 33–46. DOI: 10.1145/1519065.1519071. 211, 213, 214, 217, 227, 256, 257

J. Salwan. 2011. ROPGadget. http://shell-storm.org/project/ROPgadget/. 136

F. Schuster. July 2015. *Securing Application Software in Modern Adversarial Settings*. Ph.D. thesis, Katholieke Universiteit Leuven. 140

F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. 2015. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *36th IEEE Symposium on Security and Privacy (S&P)*, pp. 745–762. DOI: 10.1109/SP.2015.51. 15, 20, 67, 70, 97, 140, 182, 183, 184, 185, 186, 200, 204, 211

F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. 2014. Evaluating the effectiveness of current anti-ROP defenses. In *Research in Attacks, Intrusions, and Defenses*, volume 8688 of *Lecture Notes in Computer Science*. DOI: 10.1007/978-3-319-11379-1_5. 139, 140, 177, 182, 186, 188, 209

E. J. Schwartz, T. Avgerinos, and D. Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy (S&P)*. DOI: 10.1109/SP.2010.26. 86

E. J. Schwartz, T. Avgerinos, and D. Brumley. 2011. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Conference on Security (SEC)*, pp. 25–25. 136

C. Segulja and T. S. Abdelrahman. 2014. What is the cost of weak determinism? In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pp. 99–112. DOI: 10.1145/2628071.2628099. 254

J. Seibert, H. Okhravi, and E. Söderström. 2014. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, pp. 54–65. DOI: 10.1145/2660267.2660309. 141, 182

K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pp. 309–318. 82, 84, 173, 178

F. J. Serna. 2012. CVE-2012-0769, the case of the perfect info leak. http://media.blackhat.com/bh-us-12/Briefings/Serna/BH_US_12_Serna_Leak_Era_Slides.pdf. 63, 117

H. Shacham. 2007. 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)*, pp. 552–561. DOI: 10.1145/1315245.1315313. 62, 172, 184, 185, 186, 200, 233

H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, pp. 298–307. DOI: 10.1145/1030083.1030124. 62

N. Shavit and D. Touitou. 1995. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 204–213. 28

K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. 2013. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *34th IEEE Symposium on Security and Privacy (S&P)*, pp. 574–588. DOI: 10.1109/SP.2013.45. 10, 20, 49, 63, 82, 86, 117, 141, 177, 182, 184, 186, 203

K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis. 2016. Return to the zombie gadgets: Undermining destructive code reads via code inference attacks. In *37th IEEE Symposium on Security and Privacy (S&P)*, pp. 954–968. DOI: 10.1109/SP.2016.61. 70

Solar Designer. 1997a. "return-to-libc" attack. Bugtraq. 203

Solar Designer.1997b. lpr LIBC RETURN exploit. http://insecure.org/sploits/linux.libc.return.lpr.sploit.html. 203

C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski. 2015. Exploiting and protecting dynamic code generation. In *Network and Distributed System Security Symposium (NDSS)*. 49, 58, 60

A. Sotirov. 2007. Heap feng shui in JavaScript. *In Proceedings of Black Hat Europe*. 132

E. H. Spafford. January 1989. The internet worm program: An analysis. *SIGCOMM Comput. Commun. Rev.*, 19 (1): 17–57. ISSN 0146-4833. DOI: 10.1145/66093.66095. 61

SPARC. SPARC V8 processor. http://www.sparc.org. 206

R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. 2009. Breaking the memory secrecy assumption. In *2nd European Workshop on System Security (EUROSEC)*, pp. 1–8. DOI: 10.1145/1519144.1519145. 63, 117

D. Sullivan, O. Arias, L. Davi, P. Larsen, A.-R. Sadeghi, and Y. Jin. 2016. Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity. In *IEEE/ACM Design Automation Conference (DAC)*, pp. 83.2:1–6. DOI: 10.1145/2897937.2898098. 209

L. Szekeres, M. Payer, T. Wei, and D. Song. 2013. SoK: Eternal war in memory. In *Proceedings International Symposium on Security and Privacy (S&P)*. DOI: 10.1109/SP.2013.13. 2, 61, 82, 85, 211

L. Szekeres, M. Payer, L. Wei, D. Song, and R. Sekar. 2014. Eternal war in memory. *IEEE Security and Privacy Magazine*. DOI: 10.1109/MSP.2013.47. 2

A. Tang, S. Sethumadhavan, and S. Stolfo. 2015. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *ACM Conference on Computer and Communications Security (CCS)*, pp. 256–267. DOI: 10.1145/2810103.2813685. 70

C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. 2014. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium*. http://dl.acm.org/citation.cfm?id=2671225.2671285. 58, 86, 173, 175, 182, 204, 208, 211, 233

M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. 2011. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID)*, pp. 121–141. DOI: 10.1007/978-3-642-23644-0_7. 117, 140, 183, 184, 185, 200, 204

A. van de Ven. August 2004. New security enhancements in Red Hat Enterprise Linux v.3, update 3. http://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf. 9, 82

V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. 2015. PathArmor: Practical ROP protection using context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 927–940. DOI: 10.1145/2810103.2813673. 14, 137

V. van der Veen, N. D. Sharma, L. Cavallaro, and H. Bos. 2012. Memory errors: The past, the present, and the future. In *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, pp. 86–106. DOI: 10.1007/978-3-642-33338-5_5. 61

S. Volckaert. 2015. Advanced Techniques for multi-variant execution. Ph.D. thesis, Ghent University. 226, 231

S. Volckaert, B. Coppens, and B. De Sutter. 2015. Cloning your gadgets: Complete ROP attack immunity with multi-variant execution. *IEEE Trans. on Dependable and Secure Computing*, 13 (4): 437–450. DOI: 10.1109/TDSC.2015.2411254. 211, 250

S. Volckaert, B. Coppens, B. De Sutter, K. De Bosschere, P. Larsen, and M. Franz. 2017. Taming parallelism in a multi-variant execution environment. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pp. 270–285. DOI: 10.1145/3064176.3064178. 230, 232

S. Volckaert, B. Coppens, A. Voulimeneas, A. Homescu, P. Larsen, B. De Sutter, and M. Franz. 2016. Secure and efficient application monitoring and replication. In *USENIX Annual Technical Conference (ATC)*, pp. 167–179. 214, 215, 247

S. Volckaert, B. De Sutter, T. De Baets, and K. De Bosschere. 2013. GHUMVEE: Efficient, effective, and flexible replication. In *5th International Symposium on Foundations and Practice of Security (FPS)*, pp. 261–277. 214, 217, 232

R. Wahbe, S. Lucco, T. Anderson, and S. Graham. 1993. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pp. 203–216. DOI: 10.1145/168619.168635. 8, 9, 41, 68, 249

Z. Wang and X. Jiang. 2010. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pp. 380–395. DOI: 10.1109/SP.2010.30. 58, 208

R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. 2012. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 157–168. DOI: 10.1145/2382196.2382216. 11, 173, 177

R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. 2010. Capsicum: Practical capabilities for UNIX. In *19th USENIX Security Symposium*, pp. 29–46. 16

T. Wei, T. Wang, L. Duan, and J. Luo. 2011. INSeRT: Protect dynamic code generation against spraying. In *International Conference on Information Science and Technology (ICIST)*, pp. 323–328. DOI: 10.1109/ICIST.2011.5765261. 59

J. Werner, G. Baltas, R. Dallara, N. Otternes, K. Snow, F. Monrose, and M. Polychronakis. 2016. No-execute-after-read: Preventing code disclosure in commodity software. In *11th ACM Symposium on Information, Computer, and Communications Security (ASIACCS)*, pp. 35–46. DOI: 10.1145/2897845.2897891. 70

J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen. 2011. RIPE: Runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pp. 41–50. DOI: 10.1145/2076732.2076739. 109, 239

R. Wojtczuk. 1998. Defeating Solar Designer's non-executable stack patch. http://insecure.org/sploits/non-executable.stack.problems.html. 20, 81, 82, 203

C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. 2002. Linux security modules: General security support for the Linux kernel. In *Proceedings 11th USENIX Security Symposium*. 16

R. Wu, P. Chen, B. Mao, and L. Xie. 2012. RIM: A method to defend from JIT spraying attack. In *7th International Conference on Availability, Reliability, and Security (ARES)*, pp. 143–148. DOI: 10.1109/ARES.2012.11. 59

Y. Xia, Y. Liu, H. Chen, and B. Zang. 2012. CFIMon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, pp. 1–12. DOI: 10.1109/DSN.2012.6263958. 173, 176

F. Yao, J. Chen, and G. Venkataramani. 2013. JOP-alarm: Detecting jump-oriented programming-based anomalies in applications. In *IEEE 31st International Conference on Computer Design (ICCD)*, pp. 467–470. DOI: 10.1109/ICCD.2013.6657084. 209

B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. 2009. Native client: A sandbox for portable, untrusted x86 native code. In *30th IEEE Symposium on Security and Privacy (S&P)*, pp. 79–93. DOI: 10.1109/SP.2009.25. 8, 39, 86

B. Zeng, G. Tan, and Ú. Erlingsson. 2013. Strato: A retargetable framework for low-level inlined-reference monitors. In *USENIX Security Symposium*, pp. 369–382. 58, 86

B. Zeng, G. Tan, and G. Morrisett. 2011. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 29–40. DOI: 10.1145/2046707.2046713. 58, 59, 86

C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song. 2015. VTint: Defending virtual function tables' integrity. In *Symposium on Network and Distributed System Security (NDSS)*. DOI: 10.14722/ndss.2015.23099 . 160, 173, 176, 182

C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. 2013. Practical control flow integrity and randomization for binary executables. In *34th IEEE Symposium on Security and Privacy (S&P)*, pp. 559–573. DOI: 10.1109/SP.2013.44. 38, 82, 86, 97, 110, 114, 117, 118, 119, 127, 136, 169, 173, 174, 182, 208, 209

M. Zhang and R. Sekar. 2013. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium*, pp. 337–352. http://dl.acm.org/citation.cfm?id=2534766.2534796. 38, 82, 86, 97, 110, 114, 117, 118, 119, 127, 136, 173, 174, 182, 208, 209

H. W. Zhou, X. Wu, W. C. Shi, J. H. Yuan, and B. Liang. 2014. HDROP: Detecting ROP attacks using performance monitoring counters. In *Information Security Practice and Experience*, pp. 172–186. Springer International Publishing. DOI: 10.1007/978-3-319-06320-1_14. 173, 177

X. Zhou, K. Lu, X. Wang, and . Li. 2012. Exploiting parallelism in deterministic shared memory multiprocessing. *Journal of Parallel and Distributed Computing*, 72(5):716–727. DOI: 10.1016/j.jpdc.2012.02.008. 230