GHUMVEE: Efficient, Effective, and Flexible Replication

Stijn Volckaert*, Bjorn De Sutter, Tim De Baets, and Koen De Bosschere

Computer Systems Lab, Ghent University
{stijn.volckaert,bjorn.desutter,koen.debosschere}@elis.ugent.be

Abstract. We present GHUMVEE, a multi-variant execution engine for software intrusion detection. GHUMVEE transparently executes and monitors diversified replicae of processes to thwart attacks relying on a predictable, single data layout. Unlike existing tools, GHUMVEE's interventions in the process' execution are not limited to system call invocations. Because of that design decision, GHUMVEE can handle complex, multi-threaded real-life programs that display non-deterministic behavior as a result of non-deterministic thread scheduling and as a result of pointer-value dependent behavior. This capability is demonstrated on GUI programs from the Gnome and KDE desktop environments.

Keywords: Memory Exploits, Non-determinism, Diversified Process Replicae.

1 Introduction

Memory error exploits divert the control [2] or data flow [10] of a vulnerable program by injecting faulty data. This is typically done by overwriting data such as code pointers. Examples of such exploits are stack-smashing [2], return-oriented programming [30], and return-to-lib(c) attacks [27]. Such attacks nearly always rely on knowledge about the memory layout of the attacked application.

Several protection strategies exist to fix the vulnerabilities [5, 40], to protect against buffer overflows at run-time [1, 12, 44] to protect against the execution of injected code [22], and to prevent the attacker from determining the addresses of data [29]. Modern OSes and system libraries support all of these approaches to *prevent intrusions* and to *prevent damage* in case of intrusions. For example, the Linux and glibc support Address Space Layout Randomization (ASLR) [29] and Exec Shield [26] to prevent code on the stack to be executed, and length-bounded string functions like `strncpy` and `strncat` [38]. Extensions have been proposed in academics, such as Address Space Layout Permutation (ASLP) [20].

Many protections have been circumvented, however. Return-to-lib(c) [27] and return-oriented programming [30] attacks simply do not require injected code to be executed, the use of more secure library functionality like `strncpy` has proven to be error-prone [25] and ASLR was attacked in a brute-force manner [37].

* The authors want to thank the Agency for Innovation by Science and Technology in Flanders (IWT) and Ghent University for their support.

J. Garcia-Alfaro et al. (Eds.): FPS 2012, LNCS 7743, pp. 261–277, 2013.

A more reliable protection based on *intrusion detection* was proposed in 2006. Cox et al [14] implemented a Linux kernel extension to transparently run multiple diversified replicae of the same application in parallel. The protection relies on the assumption that it is much harder for an attacker to make diversified replicae perform the exact same malicious behavior than it is to exploit vulnerabilities in a single application version. The replicae are executed in lock-step and are always transparently fed the exact same input. A monitor module compares the invoked output operations of the replicae before executing them. When the monitor detects any discrepancies, it assumes that those result from an attack taking place and it terminates the execution before any damage is done.

Since Cox et al, a number of other so-called multi-variant execution engines (MVEE) have been developed [8, 9, 32–36], as well as different methods to diversify applications, including stacks growing in opposing directions [32], heap layout randomization [6], redundant data diversity [28], address space partitioning [8], ASLR [29, 37], and code diversification [3, 4, 41].

However, a major problem of all existing MVEEs is that they cannot handle real desktop applications. The fundamental reason is that real-world applications are not deterministic because of non-deterministic thread scheduling and because their behavior depends on concrete pointer values, which vary when replicae are diversified. By contrast, pre-existing MVEEs and their diversification schemes only function on simple, single-threaded applications. Moreover, their memory layout diversification is limited to relatively weak, predictable forms.

This paper presents the Ghent University MVEE or GHUMVEE. Contrary to the existing MVEEs, GHUMVEE's design supports a wide range of features observed in non-deterministic applications and a wider range of diversification techniques, including the stronger protection of the less predictable, full ASLR. Furthermore, experiments demonstrate that GHUMVEE comes with less performance overhead than existing MVEEs. The most fundamental novel aspect of GHUMVEE's design is its ability to intervene in the execution of replicae at program points other than system calls. This does require some cooperation of the application developer, but as we will discuss in the paper, compilers can easily limit the burden on the developer.

Section 2 discusses related work and the weaknesses of existing MVEE's that GHUMVEE's design overcomes. Section 3 presents this design, which is evaluated in Section 4. Section 5 draws conclusions.

2 Related Work

Software memory exploit techniques and countermeasures have been actively researched in the past 15 years. Stack overflow attacks have long been the easiest way to seize control of a running application. Smashing [2] the stack allows for an attacker to inject shell code or overwrite return addresses [27]. Several solutions were proposed to eliminate stack overflow attacks. StackGuard [12] inserts canaries into the stack to detect return address overwrites. Several state-of-the-art compilers adapted this technique later on [19, 24]. Other people proposed

the use of a secondary stack to keep copies of the return addresses [11, 21]. An alternative is to extend the C-library functions that are commonly used to set up the attack, with extra security checks such as bounds checking [1, 44]. LibSafe [5, 40] does this at runtime and modern versions of the GCC and VC++ compilers offer alternative versions of these functions [16, 23]. Mainstream operating systems also implement some form of software-enforced Data Execution Prevention [22, 26] to prevent injected code from being executed. Most other exploiting techniques, control-data attacks in particular, share one important property. They all make assumptions about the memory layout of the target application, e.g., about the absolute locations of certain functions or about the distance between two allocated objects. Many proposed techniques attempt to break these assumptions, all of which involve randomization. Xu [43] modified the Linux program loader to dynamically relocate a program's stack, heap and shared libraries. The PaX team implemented and demonstrated Address Space Layout Randomization [29], a well known technique with goals similar to the latter. ASLR employs a kernel patch to relocate a program's stack, heap and shared libraries during startup. All mainstream operating systems have adapted ASLR. Other techniques exist but are not as commonly used [13, 20].

In 2005, Berger and Zorn proposed DieHard [6], a framework for redundant execution of multiple diversified program replicae. DieHard tries to protect programs against memory errors and exploits thereof by running multiple replicae of the same program in parallel and feeding them the same input. A custom memory allocator ensures full heap randomization of the replicae: objects always have different addresses in the different replicae. DieHard redirects all program output through `stdout` to its voter module where it can isolate replicae that encounter memory errors. DieHard does not require any kernel modifications but can only run replicae whose only input comes from `stdin`, it cannot run multi-threaded programs or any other programs with pointer-dependent behavior.

More advanced MVEEs are the N-Variant Systems [14, 28], the proof-of-concept MVEE from Cavallaro et al [8, 9] (hereafter called CPoC), and Orchestra [32–36]. Figure 1 displays their basic operation. The kernel or user-space monitor is responsible for running multiple replicae of a process in a user-transparent manner. To that extent, the monitor intercepts all communication between the replicae and the outside world. As the progress of the replicae (denoted by black bars on the horizontal time access) may differ, they will communicate through system calls at different moments in time. The monitor intercepts the calls, stalls the calling replicae and waits until all of them have made a call. At that time, the monitor compares the calls and operands, and either terminates the program or handles the calls appropriately. For example, when `sys_brk` is invoked to request memory from the OS, the monitor checks the requested sizes and lets the OS allocate memory for both replicae after which both replicae continue executing. When `sys_write` is invoked to write to a file, the monitor blocks one of them to ensure transparency for the user. The result of the system call is still fed back to both replicae, which continue their execution.

The first one concerns the *rendez-vous points* at which MVEEs intervene in the execution of the replicated program versions. The aforementioned MVEEs only intervene in a replica to intercept system calls. By construction, those MVEEs can therefore not handle any non-trivial multi-threaded program. The reason is that the threads in the different replicae have to be executed in the same order (i.e., synchronized) to avoid false alerts. No fully deterministic execution is required (i.e., consecutive runs of the application under control of an MVEE may feature different thread schedules), but within one execution under control of an MVEE, all synchronization events and decisions need to be replicated. For example, when a program allocates tasks in a pool to spawn task threads, they need to be spawned in the same order in all replicae. And when the tasks updates shared memory, that needs to happen in the same order in all replicae. MVEEs that only intercept applications upon system calls cannot provide this synchronization for two reasons. First, many modern applications feature synchronization operations that do not involve system calls. These include atomic functions, (uncontended) locks by means of futexes [17], and many custom synchronization primitives. Secondly, several synchronization primitives such as the pthread_cond_timedwait execute multiple system calls as part of a more abstract decision process. To replicate these decisions, a replication mechanism at a higher abstraction level than system calls is needed as well. As will be discussed in detail in Section 3.3, GHUMVEE supports such a replication mechanism that solves both issues with acceptable performance overhead.

The second limitation of existing MVEEs is their lack of support for program behavior that depends on exact pointer values. N-Variant Systems and CPoC rely on address space partitioning, in which each concrete address that can be targeted by an attack occurs in only one replica. Orchestra features two replicae in which the stack grows in different directions to prevent a buffer overflow from having the same effect in both replicae.

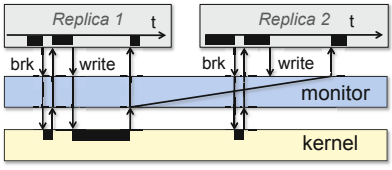


Fig. 1. Basic operation of a MVEE

The problem with these address-space based approaches is that many applications' behavior depends on concrete pointer values. Those values are then typically hashed to index hash tables or other containers such as (supposedly unordered) sets. When the computed hashes differ in the different replicae, their behavior diverges in many ways. For example, when collusions in a hash table differ in two replica, they might rehash or resize the hash tables at different points in time. In the case of a resize, this might involve memory allocation system calls being executed in one replica but not in the other. None of the existing MVEEs can handle this. When iterating over supposedly unordered containers in which objects are stored based on hashed pointer values, the order will also depend on the concrete values. So the order of visiting objects and performing tasks on them might differ from one replica to the other. In some cases the tasks involve no sensitive operations, but in many cases they do. This ranges from

different files being opened for different objects, over locks being taken on the visited objects, to worker threads being spawned for the stored objects.

In other words, if the order in which objects are stored in containers is not controlled by the MVEE, different replicae may show diverging behavior in every possible way. All existing MVEEs that we are aware of suffer from this problem. This has two consequences. First, they are applicable only to relatively simple, nice programs. For example, the programs evaluated in the existing MVEE papers are limited to a modified Apache, thttpd, SPEC benchmarks, and Snort. Exactly how nice the programs have to be depends on the precise details of the MVEE internals. Secondly, because of this dependence on different aspects of nice behavior, the existing MVEEs provide only relatively weak forms of protection. Orchestra is limited to protecting buffer overflows on the stack. The partitioned address spaces of N-Variant Systems and CPoC ore a very limited form of layout diversification with very predictable behavior.

As discussed in Section 3.7, GHUMVEE can handle many modern programs with address-dependent behavior, none of which are handled correctly by pre-existing MVEEs. Moreover, GHUMVEE can replicate the applications with full code and data layout randomization. As such, GHUMVEE makes the protection provided by replicated execution applicable to a much wider set of applications, and it demonstrates that stronger forms of protection can be provided. These are the two most important contributions of this paper.

3 GHUMVEE Architecture

The GHUMVEE monitor is launched from the command line with the program to be protected as its argument. From a database GHUMVEE then retrieves the executables of the replicae. GHUMVEE spawns the replica processes to which it attaches itself using Linux' `ptrace` API [39]. From then on, GHUMVEE acts as a proxy between the replicae and the kernel as depicted in Figure 1.

3.1 Rendez-Vous Points

GHUMVEE can intercept all system calls invoked by the replicae and manipulate or stall them when needed. GHUMVEE's rendez-vous points are system call entries (i.e, invocations) and exits (i.e., returns). Replicae are stalled at both types of points and not resumed until all replicae have reached the rendez-vous point. GHUMVEE handles rendez-vous points based on the type of system call the replicae are trying to execute. We generally distinguish four types of system calls. The distinction is based on four factors:

I/O-Related System Calls: These system calls should be performed only once to ensure transparency and to avoid unwanted side effects. For example, when replicae are writing to a file, the data should be written only once, precisely like it would happen in the original program.

Side Effects: System calls that create, modify or delete process-bound kernel structures have side effects. Most memory management functions are examples of such system calls. These calls are performed by all replicae in the same manner as depicted in Figure 1.

Mutable Results: System calls that have mutable results, i.e., calls that return different results upon every invocation, should only be performed once to ensure that all replicae get consistent return data from these calls. Most time-related functions are examples of such calls.

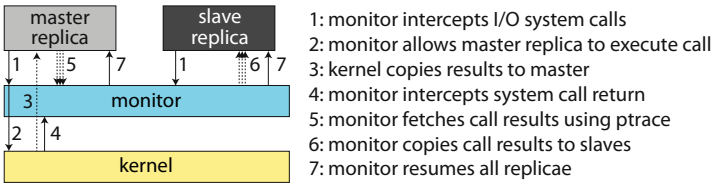
Self-Aware: System calls that make a process self-aware should only be performed once. These include `sys_getpid` and `sys_open(/proc/self/...)`.

After a system call entrance has been handled in accordance with the call's class, the monitor waits for the replicae to hit the next rendez-vous point. In most cases, this is the system call's exit. Handling this rendez-vous point is straightforward. If the system call was executed by all replicae, the monitor checks whether all of them received consistent results from the call. Then it either resumes them or shuts down the system, e.g., if a call returned an error for some replica but not for the others. If on the other hand, the call was only executed by one replica, the monitor copies the return data into the address spaces of the slave replicae, after which it resumes all of them.

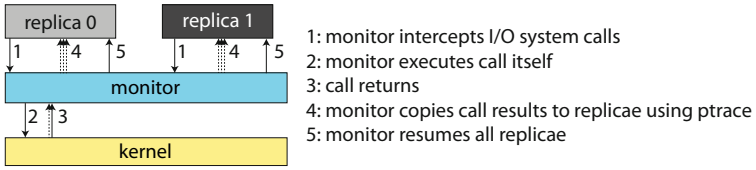
3.2 I/O Replication and Data Transfers

As mentioned above, MVEEs generally allow I/O related system calls to be executed only once. Nearly every existing solution deals with this restriction differently. Cox et al [14] stall all slave replicae in kernel-space while the master replica executes the actual I/O call. When that call returns, the monitor copies its return value and return data to the address spaces of the slave replicae.

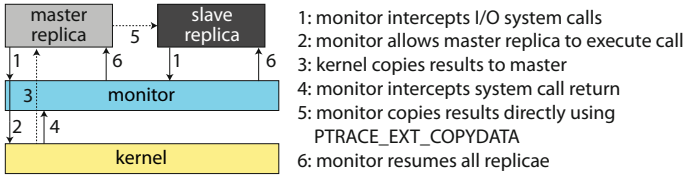
Later MVEEs handle I/O replication in user-space using Linux debugging facilities such as the `ptrace` and `waitpid` APIs [39]. The `ptrace` API allows for a debugger to observe and control the execution of a debuggee process by inspecting and manipulating its internal state, while `waitpid` is used to poll a debuggee for status changes such as the entrance or exit of a system call. CPoC's [9] implementation is similar to Cox'. The fact that CPoC stalls the replicae in user-space does entail an additional issue, however. When a process (e.g., a slave replica) is stalled at the entrance of a system call, that process cannot be prevented from executing the actual call once it is resumed. To skip such as system call, a debugger has to replace its number in register EAX by that of another system call that has no side effects. The best choice for this purpose is `sys_getpid`. After replacing the system call number and resuming the replaced slave calls, CPoC waits until the master replica returns from the original system call and until the slaves returns from their fake `sys_getpid` calls. CPoC then first copies the results of the master system call from from master replica to the monitor, and then from the monitor to all slave replicae. This process is visualized in Figure 2(a), in which solid arrows denote control transfers and dashed arrows denote data being copied.



(a) CPoC's I/O replication



(b) Orchestra's basic I/O replication



(c) GHUMVEE's I/O replication

Fig. 2. I/O replication in three MVEEs

Salamat [34] implemented a different system in Orchestra. Rather than letting a master replica execute a system call, Orchestra executes the call on behalf of the variants and copies the results of the call directly from the monitor to the replicae. This is visualized in Figure 2(b).

GHUMVEE's implementation of I/O replication is nearly identical to CPoC's. Like CPoC, we only allow the master replica to execute the original system call. Unlike CPoC however, we often do not copy the results of the system call from the master to the monitor. Instead, we copy the results directly from the master to the slaves as shown in Figure 2(b). As a result, GHUMVEE performs one less memory copy operation per replicated I/O call than CPoC and Orchestra.

In Figure 2, the copying between monitor and replicae is depicted with multiple arrows. This reflects the limitation of copying only one memory word at a time with the `PTRACE_PEEKDATA` and `PTRACE_POKEDATA` operations. On the x86 architecture, this implies that at most 4 bytes can be copied per peek or poke, each of which requires the monitor to perform a `ptrace` system call. Even in the simplest applications, this introduces a significant performance penalty.

Salamat [34] proposed a workaround that consists of shared memory buffers between the monitor and the replicae, the standard `memcpy` to copy data between that shared memory and the monitor's private memory, and a custom `memcpy`

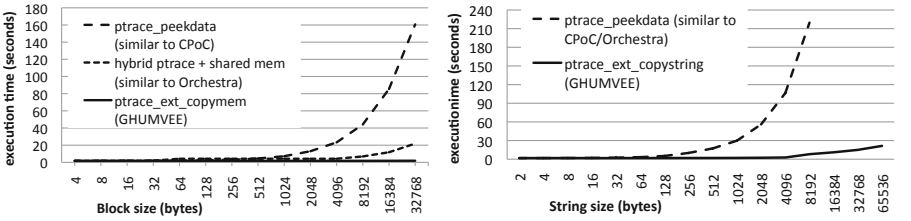


Fig. 3. Comparison of different data transfer methods

function injected into each replica. For every transfer of 40 bytes or more, control in the replicae is diverted to the injected functions to transfer data from the shared buffers to the replicae's private memories. So every transfer requires two copies: one into the shared memory and one out of it. A similar overhead would exist when `/proc/<pid>/mem` would be used instead, as that cannot be mapped directly into a process' address space.

GHUMVEE avoids part of this overhead with two small extensions for the `ptrace` API in Linux [39]. The `PTRACE_EXT_COPYDATA` and `PTRACE_EXT_COPYSTRING` operations enable a monitor or a debugger to copy a fixed-size data block and a NULL-terminated string directly to, from, and between any of its replicae or debuggees. GHUMVEE uses these extensions for all data transfer operations when it finds them in the kernel, hence the horizontal arrows in Figure 2(c). On synthetic performance benchmarks that stress the data transfer functionality of our MVEE, we obtained the results depicted in Figure 3. This figure shows that GHUMVEE's optional kernel extensions allow for data to be transferred much more efficiently. In real-world benchmarks such as SPEC CPU 2006, these extensions improved multi-variant performance by 1 to 4%.

3.3 Multi-threading and Synchronization

Arguably the biggest challenge for a MVEE is to deal with multi-threaded replicae. This is complicated mainly because MVEEs running in user-space cannot control the order in which threads are scheduled[1]. This implies (1) that a monitor can observe system calls in different orders in multi-threaded replicae because of different progress rates and different scheduling of the replicae, and (2) that the replicae of programs with non-deterministic behavior can actually perform different system calls in different replicae.

The first problem can be solved easily with a multi-threaded monitor. Like the other MVEEs that support fork/exec and multi-threading, the GHUMVEE monitor spawns a new monitor thread for every set of new processes or threads spawned by the replicae. This works fine as long as the replicae behave deterministically and execute in lock-step, because then they will spawn the same processes and threads from within the same processes and threads. Each new

[1] And even when the monitor would run in kernel-space, if has no direct control over user-space synchronization events, so the fundamental problems remain the same.

monitor thread then attaches to the corresponding replicae threads, after which each such monitor thread observes only the system calls in those corresponding threads, which will happen in exactly the same order in all replicae.

The second problem is much harder to solve. Pre-existing MVEEs simply neglect this and are hence broken for many applications, for which they report mismatches between the replicae and halt the execution. Fundamentally, the problem is that any synchronization race in non-deterministic programs can lead to different replicae executing different system calls in different orders.

In GHUMVEE, we solved this problem by forcing all slave replicae to behave exactly like the master. Whenever a synchronization race in the master replica is decided, that same decision is imposed onto the slave. This is similar to techniques used for record/replay of multi-threaded applications [31], the difference being that in GHUMVEE all replicae run concurrently in lock step, rather than sequentially. We therefore don't have to store logs of the synchronization events. Please note that GHUMVEE does not eliminate non-determinism. Rather, it only forces all replicae to take the same decision for every synchronization race. This way, GHUMVEE cannot introduce any deadlocks in the replicae.

Initially, we interposed [15] or detoured [18] all user-space synchronization operations by means of fake system calls through which the monitor became aware of the operations for which it could then enforce scheduling decisions. This solution introduced too much overhead, however. Even simple applications like the `gcalctool` calculator from the Gnome desktop environment spawn several threads during their initialization, in which they perform mostly uncontended synchronization. For example, we observed `gcalctool` performing 1.8M futex operations during its 400 ms initialization. Interposing all those operations with a fake system call and multiple ptrace system calls made the initialization time grow to over 370 seconds, a slowdown with a factor 925!

As an alternative, we designed a system with which the replicae can synchronize themselves. This is visualized in Figure 4. When the monitor spawns the replicae, it allocates shared circular buffers (shown in green) between them. Furthermore, the monitor preloads a dynamic library with interposers and detours to intervene in all user-space synchronization events. Instead of executing system calls in all replicae as in our initial solution, these new interposers record the synchronization decisions of the master replica (e.g., the order in which its threads acquire locks) in the shared buffers. In the figure, the master threads record the order in which they acquired a specific lock Lm in the buffer. In the slave replicae, the interposers read these decisions, and impose the same behavior on the slaves. In the figure, when slave thread Bs first tries to acquire the corresponding lock Ls, the interposer observes that thread As should acquire it first, so it blocks thread Bs. When thread As tries to acquire the lock, this succeeds, and after it is released, thread Bs will acquire it as well.

As all interposers perform their duties without additional system calls or context switches to the monitor, the overhead of this solution is much smaller. For example, with this solution the already mentioned `gcalctool` initializes in 1.7 seconds, a slowdown with factor 4.25. This is still significant, but most of it

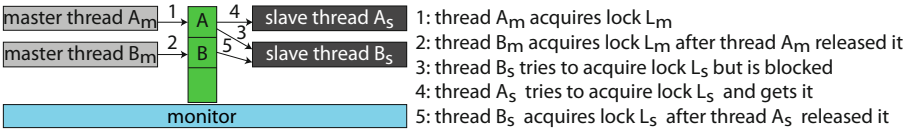


Fig. 4. GHUMVEE's synchronization decisions through shared buffers

is due to setting up the shared buffers. After the initialization the rate at which synchronization operations are performed decreases significantly, as a result of which this enforced synchronization between different replicae does not results in an noticeable overhead during the normal, interactive operation of the program.

Enforcing the master's schedule on the slave replicae in this way solves non-determinism problems related to synchronization races, a form of non-determinism that is generally considered acceptable program behavior and that occurs in most modern multi-threaded programs. Our solution does not protect against non-determinism caused by critical data races. As such data races are generally considered as bugs, we feel this is an acceptable limitation of GHUMVEE.

3.4 Signal Handling

Besides control-data dependencies in multi-threaded applications, there are several other sources of non-determinism. One of them is asynchronous signal delivery. Because most signal handlers invoke system calls, delivering signals from external sources to replicae should happen very carefully. For example, assume that one single-threaded replica is blocked on entry to a system call, waiting for the other single-threaded replicae to arrive at the same point. If we then deliver a signal to another replica that is still executing, the corresponding signal handler in that replica will be invoked, in which a very different system call might be invoked in turn, leaving two replicae wanting to execute different system calls.

In GHUMVEE, this is solved by delaying the delivery of signals to replicae until they are blocked on exit of a system call. This can significantly delay the handling of a signal. Salamat et al proposed a complex solution to deliver signals earlier [34], which showed significant improvements for synthetic signal handling stress tests. We investigated the need for such a complex solution for real-world applications, and discovered that in real applications, the number of signals is typically more than three orders of magnitude lower than the number of system calls invoked. As such, limiting the delivery of signals to the rendezvous points of those system calls does not hinder performance or latency in practice. One notable side effect of our signal handling mechanism is that some duplicate asynchronous signals might be lost. In practice however, we have not encountered any programs that started behaving incorrectly when a duplicate signal was not delivered.

Unlike asynchronous signals, synchronous signals are delivered immediately. Synchronous signals occur as a direct result of the executing instruction. Because the MVEE keeps all replicae in a consistent state, we can assume that all replicae will trigger the same synchronous signal on the same instruction.

3.5 Time Stamp Counter

Yet another source of non-determinism is time, of which applications can become aware through the `gettimeofday` system call. On the x86 architecture, the time can also obtained directly from the processor by executing the `rdtsc` instruction in user mode. Salamat [34] acknowledge the problems this can cause when different replicae get different input through `rdtsc`, but he offers no solution beyond pointing out that programmers could use `gettimeofday` instead. In practice, programmers do not follow his advice, however. A simple program like the `gcalctool` calculator executes the `rdtsc` instruction tens of times.

GHUMVEE solves this by setting the control registers in the x86 architecture to make all user-mode `rdtsc` instructions trap. The monitor handles the resulting `SIGSEGV` signal by feeding all replicae the same time stamp counter value.

3.6 Shared Memory Support

Linux programs can use shared memory blocks to set up communication channels with other programs. Once such channels are in place, the programs can communicate by reading and writing from and to the shared memory without using any system calls and without exchanging any other information. This makes it very hard for a MVEE to perform correct replication under all circumstances.

Somewhat surprisingly, almost all programs use shared memory. So at least partial support is needed in an MVEE. But fortunately, most programs do not really require two-way communication channels with the outside world via shared memory. An analysis of the usage of shared memory in our testing applications reveals that shared memory is typically used for one of the following goals:

Shared Libraries: The Linux program loader uses shared memory blocks to map shared libraries into the address spaces of dynamically linked programs. This should obviously be supported by an MVEE.

Memory-Mapped I/O: When a file is mapped into a program's address space as shared memory, it can be read and written without the overhead of system calls. We have encountered several programs in the KDE desktop environments that require memory-mapped I/O to start up properly.

Internal Communication: Anonymous shared memory is not accessible to external programs. We have encountered several multi-threaded programs that used anonymous shared memory to set up additional heaps, e.g., with large contiguous pages. Anonymous shared memory can only be accessed by the allocating program and its descendants. Since these descendants also run under MVEE control, all communication through anonymous shared memory can be controlled using the techniques described in Section 3.3.

Non-anonymous 2-Way Communication Channels: Several programs try to set up 2-way communication with the outside world through the System V `sys_ipc` system call. As indicated, this cannot be handled efficiently. We also discovered, however, that all programs we studied have backup schemes for when the System V call is not supported, i.e., when it

fails. That backup uses the above types of shared memory, as well as regular communication channels like pipes, signals and system calls. As those channels can be handled by MVEEs without a problem, it suffices to let the monitor intercept the requested shared memory allocation by means of the System V system calls and let them return as if the requests failed.

Several solutions have been proposed in the past to deal with the first three cases [9, 34]. GHUMVEE builds on those solutions. Although the classification above seems pretty straightforward, it is not easy to allow all safe uses of shared memory while blocking the unsafe forms. Memory-mapped I/O is particularly hard to support because memory-mapped files and regular 2-way communication channels are set up the same way. Cavallaro [9] proposed to solve this problem by using the CPU's page exception mechanism but indicated that this approach might incur a lot of overhead. For that reason we did not even consider this solution. Instead GHUMVEE supports memory-mapped I/O by manipulating the `sys_mmap` and `sys_mmap2` used to map shared memory onto files. Normally, memory-mapped files are mapped by passing the `MAP_SHARED` flag to the mmap call. Our monitor disables this flag and enables the `MAP_PRIVATE` flag instead. This way, the requested file is mapped into the address spaces of the replicae, but any changes to the file are not written back to the file when the block is unmapped. Instead the GHUMVEE monitor keeps track of these manipulated blocks and performs the write-back of the file data itself.

This approach prevents programs from using shared memory based 2-way communication channels without notifying the programs, but in practice, we have not encountered any program that stopped working because of our solution.

3.7 Address Space Layout

Finally, we observed that many real-world applications and libraries (including GTK+, Glib, Pango, KDE, and LibreOffice libraries) exhibit behavior that depends on pointer values. As explained in Section 2, the main problem with pointer values being hashed into keys to access data structures is that the data structures are resized, restructured or iterated through in orders that depend on keys obtained from hashing pointer values. As a result of these dependencies, almost all non-trivial programs we tried fail on existing MVEE's with ASLR enabled. Nonetheless, this problem is not mentioned in any MVEE-related paper.

We tackle this problem by interposing the hash functions that compute pointer-dependent keys, similarly to the way we interpose synchronization operations. In the master replica, the interposer wraps the hash function and passes the computed keys to shared queues. In the slave replicae, the interposers replace the hash functions. Instead of computing a hash key, they obtain them from the queues. That way, all replicae use the same hash keys. This solution is not fool proof, as it only works for limited uses of the hash keys, such as for indexing and ordering data structures. In more complex scenarios, e.g., where the hashing is replaced by encryption and where the encrypted keys also get decrypted, GHUMVEE can still fail. But for the applications we tested that use the aforementioned libraries, GHUMVEE works fine.

4 Experimental Evaluation

Validation. We tested GHUMVEE on numerous interactive, multi-threaded programs, including Gnome tools such as `gcalctool`, KDE tools such as `kcalc`, and LibreOffice on a quadcore Core i7 870 system running Ubuntu 11.04. For, e.g, LibreOffice we tested operations such as opening and saving files, editing various types of documents, running the spell checker, etc. In these tests GHUMVEE spawned between one and four replicae from the same executable. Tests were conducted with and without ASLR enabled. Without ASLR, all addresses occurring in the replicae are identical. With ASLR, most addresses are different in all replicae. This includes the addresses of data on the heap and on the stack, as well as addresses of statically allocated data and code in dynamically linked libraries. We also evaluated GHUMVEE on a number of SPEC benchmarks, mainly to evaluate performance and to validate GHUMVEE on replicae with code diversification. In particular, we compiled the SPEC2006 benchmark with GCC 4.5.2 at optimization levels -O2 and -O3. This allows us to test GHUMVEE on replicae in which even the static code addresses differ.

All tests succeeded. This demonstrates that GHUMVEE is more flexible than existing MVEEs, in the sense that it supports a wider range of applications, as well as a wider range of data and code diversification techniques, including full layout randomization which presents a much less predictable target to attackers.

Transparency. From a user perspective, GHUMVEE is completely transparent. Except for having to launch an application with the GHUMVEE monitor and the performance and memory consumption overhead involved with the use of GHUMVEE, there is no noticeable effect.

From a developer perspective, however, GHUMVEE is not completely transparent. In particular GHUMVEE's reliance on interposers is not fully transparent. To handle synchronization as described in Section 3.3 and address space layout differences as described in Section 3.7, someone has to implement the appropriate interposers. Besides highly application-dependent use of pointer values to index data structures, many real-world applications use custom synchronization mechanisms [42] as well as custom memory allocators [7] besides the standard `glibc` and `pthread` primitives (despite the cited literature demonstrating how bad that customization practice is). In all these cases, the application developers themselves are responsible (1) for ensuring that interposers can handle all cases correctly, and (2) for providing the interposers.

For our whole test suite, we wrote 2863 lines of interposer C code. Their distribution over different libraries is shown in Table 1. Of those 2983 lines, 2509 cover the functionality in standard libraries and the header files of the GHUMVEE interposer API. Those are readily available to all GHUMVEE users and need not be reimplemented. For specific applications (Gnome and LibreOffice) and the libraries they rely on, 474 lines of very simple C interposer code suffice. For example, defining an interposer for the hash function `gtk_gc_value_hash` that takes an argument of type `gpointer` (as defined in that library) to produce a hash of type `guint` looks as follows:

Table 1. Programming effort for GHUMVEE's interposers

standard library	interposer library (header files)		libc	pthread	interposer base lib	total
lines of C code	260		654	766	829	2509
application library	glib	gtk	orbit	pango	libreoffice	total
lines of C code	105	54	78	54	183	474

```
INTERPOSER_DETOUR_GEN_HOOKFUNC(guint, gtk_gc_value_hash, (gpointer key)) {
    MVEE_DO_SYNC(guint, (key), int, MVEE_GTK_HASH_BUFFER, 0);
    return result;
}
```

This code specifies that the slave replicae should obtain the hashed value with the width of an `int` from the master through the `MVEE_GTK_HASH_BUFFER` instead of computing a hash themselves (`0`). `INTERPOSER_DETOUR_GEN_HOOKFUNC` and `MVEE_DO_SYNC` are preprocessor macro's defined in one of the GHUMVEE source code headers. The first macro generates an empty trampoline [18] and generates detour registration code that automatically detours [18] the target function (`gtk_gc_value_hash`) when the interposer library is loaded. The second macro generates all of the synchronization code. In the master replica this synchronization code writes all computed hash values into a circular buffer that is shared between all replicae. In the slave replicae, the code reads the computed values from that same buffer.

Writing the code for other interposers is similarly mechanistic. The effort required by the developer is therefore by and large limited to identifying the functions that need to be interposed and to make them interposable. For the latter, i.e., for ensuring that interposers can handle all cases correctly, we only needed to modify 54 lines of code in the applications and their libraries to convert static and hence non-interposable functions into dynamic, non-inlined interposable ones. This only required removing the `static` keyword and `inline` attribute from the code. Additionally, the LibreOffice link-script had to be adapted to make the symbols corresponding to the hash functions visible in the linked binary. This also is a trivial edit.

More changes were needed in `glibc`, which contains a large amount of inline assembly. Besides the 654 lines of interposer code, our `glibc` patch is 845 lines long. This patch can of course be reused by all GHUMVEE users.

Altogether, this limited and mechanistic programming overhead (part of which can probably be automated by a compiler working on the basis of pragmas) demonstrates the limited burden on application programmers to make their applications compatible with GHUMVEE. Because of its reliance on widely applicable and easy to use interposers, GHUMVEE is a much more flexible tool than existing MVEEs.

Performance Overhead. To measure the performance overhead of GHUMVEE, we measured the execution times of several multi-variant combinations of SPEC

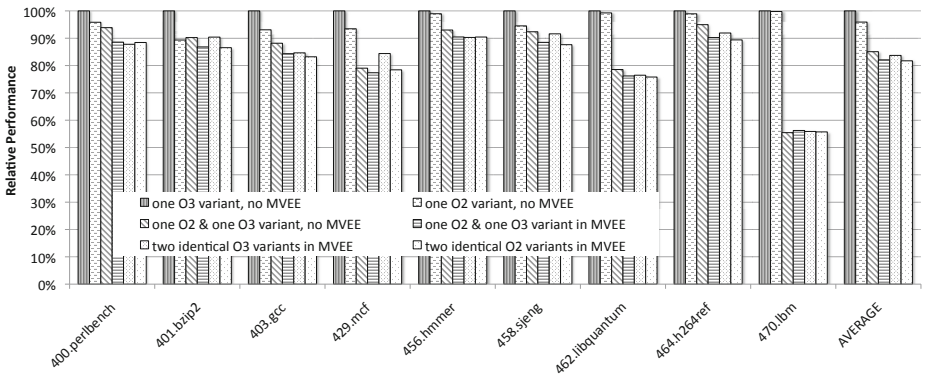


Fig. 5. Relative performance of SPEC benchmarks running under the GHUMVEE

benchmarks compiled at different optimization levels. The relative performance (i.e., the relative execution times) are depicted in Figure 5.

Comparing the results of GHUMVEE with two O3 variants (yellow bars) to those with one O3 variant (gray bars), we observe that on average GHUMVEE comes with a 16% performance penalty. For O2 variants (orange vs. blue), the performance penalty is 15%. The average overhead of GHUMVEE is hence slightly smaller than the 17% reported for Orchestra [34], despite the fact that GHUMVEE performs equivalence checks for many more system calls.

For `403.gcc`, the only benchmark common to our evaluation and that of Orchestra, the overheads are 15% and 17% for O3 and O2 resp. with GHUMVEE, and about 30% with Orchestra. Given that `403.gcc` is an I/O intensive benchmark, this difference in performance can be attributed to the kernel extension discussed in Section 3.2.

For `470.lbm`, the overhead of 46% is particularly high. This overhead is not due to the overhead of GHUMVEEs interventions, however. A similar overhead can be observed when two variants of this benchmark run side by side without an MVEE. As `470.lbm` is a memory-intensive benchmark, the shared caches and the shared memory buses on the Core i7 become the bottleneck when running multiple variants concurrently, with or without the MVEE. This demonstrates that although GHUMVEE limits the overhead of its interventions, it cannot magically reduce the inherent overhead of replicating processes that are not fit for replication due to resource contention. In this regard, GHUMVEE comes with the same limitations as any other MVEE.

5 Conclusions

We presented the Ghent University multi-variant execution engine or GHUMVEE, the first intrusion detection system based on the execution of diversified replicae that supports full ASLR and real-life applications. We presented novel techniques to support thread synchronization and pointer-dependent program behavior, as

well as other sources of non-determinism such as time stamp counters. To the extent a comparison with existing systems was possible, GHUMVEE proved to be more effective, more efficient, and more flexible than existing MVEEs.

References

1. Akritidis, P., Costa, M., et al.: Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In: Proc. USENIX SSYM, pp. 51–66 (2009)
2. Aleph One: Smashing the stack for fun and profit. Phrack Magazine 7(49) (1996)
3. Anckaert, B.: Diversity for Software Protection. PhD thesis, Ghent University (2008)
4. Anckaert, B., Jakubowski, M., Venkatesan, R.: Proteus: virtualization for diversified tamper-resistance. In: Proc. ACM DRM, pp. 47–58 (2006)
5. Baratloo, A., Singh, N., Tsai, T.: Libsafe: Protecting critical elements of stacks. White paper, Bell Labs, Lucent Technologies (December 1999)
6. Berger, E., Zorn, B.: DieHard: probabilistic memory safety for unsafe languages. In: Proc. ACM PLDI, pp. 158–168 (2006)
7. Berger, E.D., Zorn, B.G., McKinley, K.S.: Reconsidering custom memory allocation. In: Proc. ACM OOPSLA, pp. 1–12 (2002)
8. Bruschi, D., Cavallaro, L.: Diversified Process Replicæfor Defeating Memory Error Exploits. In: Proc. IEEE IPCCC, pp. 434–441 (2007)
9. Cavallaro, L.: Comprehensive Memory Error Protection via Diversity and Taint-Tracking. PhD thesis, Universita Degli Studi Di Milano (2007)
10. Chen, S., Xu, J., Sezer, E., Gauriar, P.: Non-control-data attacks are realistic threats. In: Proc. USENIX SSYM (2005)
11. Chiueh, T.C., Hsu, F.H.: RAD: A Compile-Time Solution to Buffer Overflow Attacks. In: Proc. IEEE ICDCS, pp. 409–417 (2001)
12. Cowan, C., Pu, C., et al.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: Proc. USENIX SSYM, pp. 26–29 (1998)
13. Cowan, C., Beattie, S., Johansen, J., Wagle, P.: PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities. In: Proc. USENIX SSYM, pp. 91–104 (2003)
14. Cox, B., Evans, D., et al.: N-variant systems: A secretless framework for security through diversity. In: Proc. USENIX SSYM, pp. 105–120 (2006)
15. Curry, T.W.: Profiling and Tracing Dynamic Library Usage Via Interposition. In: Proc. USENIX USTC, pp. 267–278 (1994)
16. Holtmann, M.: Secure Programming with GCC and GLibc (2008)
17. Franke, H., Russell, R., Kirkwood, M.: Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In: Proc. Ottowa Linux Symposium (2002)
18. Hunt, G., Brubacher, D.: Detours: Binary Interception of Win32 Functions. In: Proc. USENIX WINSYM (1999)
19. IBM Research: GCC extension for protecting applications from stack-smashing attacks (2005)
20. Kil, C., Jun, J., Bookholt, C., Xu, J., Ning, P.: Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In: Proc. ACSAC, pp. 339–348 (2006)
21. McGregor, J.P., Karig, D.K., Shi, Z., Lee, R.B.: A Processor Architecture Defense against Buffer Overflow Attacks (2003)
22. Microsoft Corporation: Data Execution Prevention

23. Microsoft Corporation: Security Enhancements in the CRT
24. Microsoft Corporation: Visual C++ Linker Options: /GS (Buffer Security Check) (2002)
25. Miller, T.C., de Raadt, T.: strlcpy and strlcat Consistent, Safe, String Copy and Concatenation. In: Proc. USENIX ATEC, pp. 175–178 (1999)
26. Molnar, I.: "Exec Shield", new Linux security feature
27. Nergal: The advanced return-into-lib(c) exploits. Phrack Magazine 12(58) (2001)
28. Nguyen-Tuong, A., Evans, D., Knight, J.C., Cox, B., Davidson, J.W.: Security through redundant data diversity. In: Proc. IEEE DSN, pp. 187–196 (2008)
29. PaX Team: Address Space Layout Randomization (2004)
30. Roemer, R., Buchanan, E., et al.: Return-oriented programming: Systems, languages, and applications. ACM Trans. Inf. Syst. Secur. 15, 2:1–2:34 (2012)
31. Ronsse, M., De Bosschere, K.: RecPlay: A Fully Integrated Practical Record/Replay System. ACM Trans. Comp. Sys. 17(2), 133–152 (1999)
32. Salamat, B., Gal, A., Franz, M.: Reverse stack execution in a multi-variant execution environment. In: CATARS Workshop (2008)
33. Salamat, B., Jackson, T., et al.: Orchestra: A User Space Multi-Variant Execution Environment. In: Proc. EuroSys, pp. 33–46 (2009)
34. Salamat, B.: Multi-Variant Execution: Run-Time Defense against Malicious Code Injection Attacks. PhD thesis, University of California, Irvine (2009)
35. Salamat, B., Gal, A., et al.: Multi-variant Program Execution: Using Multi-core Systems to Defuse Buffer-Overflow Vulnerabilities. In: Proc. CICIS, pp. 843–848 (2008)
36. Salamat, B., Jackson, T., et al.: Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In: Proc. EuroSys, pp. 33–46 (2009)
37. Shacham, H., Goh, E.J., Modadugu, N., Pfaff, B., Boneh, D.: On the effectiveness of address-space randomization (2004)
38. The GNU C Library: Copying and Concatenation
39. Thorvalds, L.: Linux Programmer's Manual
40. Tsai, T., Singh, N.: Libsafe 2.0: Detection of Format String Vulnerability Exploits (2001)
41. Williams, D., Hu, W., et al.: Security through Diversity: Leveraging Virtual Machine Technology. IEEE Security & Privacy 7(1), 26–33 (2009)
42. Xiong, W., Park, S., Zhang, J., Zhou, Y., Ma, Z.: Ad hoc synchronization considered harmful. In: Proc. USENIX OSDI, pp. 1–8 (2010)
43. Xu, J., Kalbarczyk, Z., Iyer, R.K.: Transparent Runtime Randomization for Security. In: Proc. SRDS 2003, pp. 260–269 (2003)
44. Younan, Y., Philippaerts, P., et al.: Paricheck: an efficient pointer arithmetic checker for C programs. In: Proc. ASIACCS, pp. 145–156 (2010)