Sharing is Caring: Secure and Efficient Shared Memory Support for MVEEs

Jonas Vinck
imec-DistriNet, KU Leuven
Belgium
jonas.vinck@kuleuven.be

Bert Abrath
Ghent University
Belgium
bert.abrath@ugent.be

Bart Coppens
Ghent University
Belgium
bart.coppens@ugent.be

Alexios Voulimeneas
imec-DistriNet, KU Leuven
Belgium
alex.voulimeneas@kuleuven.be

Bjorn De Sutter
Ghent University
Belgium
bjorn.desutter@ugent.be

Stijn Volckaert
imec-DistriNet, KU Leuven
Belgium
stijn.volckaert@kuleuven.be

Abstract

Multi-Variant Execution Environments (MVEEs) are a powerful tool for protecting legacy software against memory corruption attacks. MVEEs employ software diversity to run multiple variants of the same program in lockstep, whilst providing them with the same inputs and comparing their behavior. Well-constructed variants will behave equivalently under normal operating conditions but diverge when under attack. The MVEE detects these divergences and takes action before compromised variants can damage the host system.

Prior research has shown that multi-variant execution only works if the variants receive identical inputs. Existing MVEEs replicate inputs at the system call boundary, and therefore do not support programs that use shared-memory IPC with other processes, since shared memory pages can be read from and written to directly without system calls.

We analyzed modern applications, ranging from web servers, over media players, to browsers, and observe that they rely heavily on shared memory, in some cases for their basic functioning and in other cases for enabling more advanced functionality. It follows that modern applications cannot enjoy the security provided by MVEEs unless those MVEEs support shared-memory IPC.

This paper first identifies the requirements for supporting shared-memory IPC in an MVEE. We propose a design that involves techniques to identify and instrument accesses to shared memory pages, as well as techniques to replicate I/O through shared-memory IPC. We implemented these techniques in a prototype MVEE and report our findings through an evaluation of a range of benchmark programs. Our contributions enable the use of MVEEs on a far wider range of programs than previously supported. By overcoming one of the major remaining limitations of MVEEs, our contributions can help to bolster their real-world adoption.

1 Introduction

C and C++ have long been the languages of choice for systems programming thanks to their unique feature sets and performance characteristics. Over the years, developers all around the world have built up huge code bases without fully realizing that C/C++'s loose language specifications and lack of safety checks make even the most carefully written programs rife with undefined behavior and memory errors [33]. Hackers routinely exploit these memory errors to infiltrate systems or to force them to leak confidential information [9, 10, 22, 23, 44–47].

Software diversity is one proposed solution against memory exploits [11, 16, 29]. By randomizing code features such as the register allocation, instruction selection, data representation, or run-time memory layout, we can generate many different *variants* of a program that differ in syntax but are semantically equivalent. Memory exploits that can successfully compromise one variant often fail at compromising most, if not all, of the other variants. Software diversity is not without its flaws, however, as it does not provide any hard security guarantees and even its probabilistic protection can be undermined through information leakage attacks [40].

Multi-Variant Execution Environments (MVEEs) alleviate these shortcomings by running multiple program variants in lockstep at the granularity of system calls while feeding them with identical inputs [5, 7, 14, 20, 21, 25, 27, 28, 31, 32, 41, 49, 51, 52, 54, 55, 57, 58]. The idea is to generate *structurally asymmetrical* variants that have a high probability (of up to 100%) of reacting differently to the same exploit payloads. A monitor detects divergences in the run-time behavior, assumes that they are the result of an ongoing attack, and takes the appropriate actions. On systems with sufficient resources, MVEEs incur negligible slowdowns.

Despite their potential, few people deploy MVEEs in practice due, in part, to the non-trivial requirements they impose on protected programs. In some constrained software development life cycles (SDLCs), e.g., for safety-critical applications, programmers already need to constrain their code to meet stringent certification and fault-tolerance requirements [34]. The cost to meet additional requirements such as those imposed by MVEEs is relatively low in such cases. In non-constrained SDLCs, the MVEE-related requirements seem too high a burden at the moment. This includes soft requirements such as ease-of-use and managable compatibility issues that can be handled by recompiling code with additional compiler passes or by local code patches.

Prior research has shown that program variants only behave equivalently if they receive identical inputs [14, 41]. However, since multi-variant monitors operate at the system call boundary, they can only distribute identical inputs they receive through system calls (e.g., reads from files, pipes, or sockets). If variants receive inputs through other channels, they can run into "benign divergences", where they start performing system calls the monitor does not deem equivalent [52]. Some such benign divergences can be handled with existing techniques and hence pose soft requirements [50].

One of the channels in question is shared-memory interprocess communication (SHM IPC). After setting up a SHM IPC channel, variants can send and receive data directly without issuing system calls. This can introduce benign divergences because external processes can modify the SHM pages while the variants are reading them, thus causing the variants to read different inputs. It could also allow compromised variants to stealthily attack and compromise other variants or external processes, fully bypassing the monitor's equivalence checks. Few existing MVEEs offer any support for programs that use SHM IPC. We found that, in most cases, the MVEE just denies the program access to SHM IPC channels, often leading the program to simply shut down because it cannot function without it [52]. In cases where SHM IPC is supported, the program quickly slows down to a point where it is no longer usable. Yet, our analysis of widely used programs and libraries revealed that without efficient support for SHM IPC, MVEEs cannot protect important classes of applications. In other words, SHM IPC support is a hard requirement that was until now not met by MVEEs.

This paper reports our research to meet the requirement. By developing techniques to support SHM in MVEEs, we enable them to protect a much wider range of applications, thus overcoming one of the last remaining fundamental hard obstacles to deploying MVEEs in practice. We make the following three contributions: First, **we present an analysis of SHM usage in modern applications, which yields concrete requirements for SHM support in MVEEs.** We observe that if MVEEs cannot meet these requirements, large and important classes of applications developed today simply cannot be protected by MVEEs.

Secondly, **we present a two-layered mechanism for supporting SHM IPC in MVEEs.** The first layer leverages the MMU to trap all accesses to SHM pages. We collect information on every trapped access and forward it to the MVEE's monitor so that it can perform the attempted operation on behalf of the variants and return the results. We then augment this mechanism with a second, far more efficient, layer that relies on dynamic analysis and an in-process agent to identify and instrument instructions that access SHM ahead of time, and to vet and replicate these accesses directly in user space, bypassing the monitor's replication facilities.

Thirdly, **we evaluate a prototype implementation** on microbenchmarks, server benchmarks, as well as a media application to show that our solution supports SHM IPC with **acceptable run-time overhead. We are the first to offer full support for SHM IPC in a state-of-the-art MVEE.**

With these contributions, we do **not** aim at improving the security provided by MVEEs. Instead, these contributions overcome a fundamental limitation of existing MVEEs, thus enabling the delivery of their existing security provisions to modern applications.

2 Background

Software Diversity. The security guarantees of an MVEE depend entirely on how they generate variants. Typically, MVEEs use *structurally asymmetrical* variants that have a high probability of responding differently to the same exploit payload. Cox et al., for example, proposed to use compile-time transformations to generate variant binaries with non-overlapping address spaces [14]. This setup thwarts any code-reuse or data-oriented attacks that rely on payloads containing absolute memory addresses. Several other MVEEs achieve the same results by relying on run-time transformations to generate asymmetrical variants of a single binary [31, 51]. More examples of effective variant generation techniques are found in literature [27, 31, 41, 51, 53–55, 58].

Rendezvous Points. One of the key tasks of the MVEE monitor is to observe variants and detect divergences in their execution behavior. Security-oriented MVEEs do this by suspending variants when they reach predefined *rendezvous points* (RVPs). Once all variants arrive at such an RVP, the monitor checks their states for equivalence and takes a corrective action when the states do not match. If the states match, the monitor resumes the variants. The vast majority of existing MVEE systems use system call entry and return points as the RVPs. They check for state equivalence by comparing system call numbers and arguments. The underlying idea is that, since processes running on modern operating systems are confined to their own private virtual address spaces, any actions that may harm other processes must use system calls, which can be observed by the monitor [14, 41]. RVPs also play a crucial role in I/O replication. When the communicating entity (typically the leader variant) reaches the return RVP of an input operation such as a `sys_read`, the monitor records the result of that operation and replicates those results to the follower variants.

Suspending variants at every system call entry and return point can be detrimental to their run-time performance. Some MVEEs hence use a second type of RVP, so-called *relaxed RVPs*, to exempt certain system calls from the strict lockstepping requirement [21, 27, 51]. When the leader variant reaches a relaxed RVP, the monitor records its state and, potentially, system call results, but allows it to continue its execution immediately. When the follower variants reach the same relaxed RVP, the monitor uses the recorded data to perform state checking and/or I/O replication. If a follower reaches a relaxed RVP before the leader does, it will wait for the leader to reach the RVP and record the necessary data.

In security-oriented MVEEs, all potentially dangerous system calls need to be surrounded by regular RVPs. That way, the MVEE can guarantee that no harmful actions can be performed unless all variants get compromised simultaneously. MVEEs built for other purposes, such as software testing [21] and safe updates [36], can use relaxed RVPs everywhere. This allows them to tolerate expected divergences [37], while maintaining the capability of detecting unexpected divergences, though not until after the divergent actions have completed.

Security-Performance Trade-offs. Prior research explored many designs for MVEE monitor with different security-performance trade-offs. Broadly speaking, we see that there are four conflicting goals MVEEs try to meet:

- **RVP Enforcement** Variants should not be able to get past RVPs without having the monitor perform the necessary state equivalency checks, lockstep synchronization with other variants, and replication of system call results (if applicable);
- **Monitor Isolation** The variants should not be able to tamper with the monitor by corrupting its memory;
- **TCB Footprint** The monitor should have minimal impact on the size of the TCB;
- **Monitor Invocation Overhead** Invoking the monitor at an RVP should incur minimal overhead.

The original MVEE design, presented by Cox et al., executed its monitor in kernel space [14]. In-kernel monitors provide strict RVP enforcement, strong monitor isolation, and low invocation overhead, but have a substantial TCB footprint. Several later designs run the monitor as a standalone process in user space and rely on the `ptrace` API to intercept system calls [8, 41, 49]. These designs also provide strict RVP enforcement and strong memory isolation. Unlike in-kernel monitors, however, they have a minimal TCB footprint and they incur high run-time overhead since they require context switching for every RVP [21, 51]. VARAN uses selective binary rewriting to intercept system calls in user space and relies on an in-process monitor to avoid costly context switches [21]. This design provides the lowest monitor overhead, but does not isolate the monitor at all, and it allows compromised variants to bypass RVPs [51]. MvArmor [27] and MonGuard [56] enforce isolation of the in-process monitor component using hardware virtualization or isolation extensions [13]. MvArmor incurs low system call interception overhead but requires substantial additions to the TCB. MonGuard similarly incurs low overhead but relies on hardware that is not available in most commodity CPUs. ReMon is a hybrid design that combines a `ptrace`-based, cross-process monitor (CP-MON) with an efficient in-process monitor (IP-MON) [51]. ReMon's CP-MON provides strict RVP enforcement for regular RVPs, while its IP-MON implements equivalence checks and replication for relaxed RVPs. The IP-MON relies on information hiding to isolate itself, as well as the replication buffer (RB) it uses for replication, from potentially compromised variants. ReMon does, however, require minor modifications to the kernel.

Benign Divergences. A big challenge in the design and implementation of an MVEE is ensuring that the variants behave equivalently if they receive inputs from other sources than the system call interface. Prior research identified several such sources, including the CPU's time stamp counter and random number generator [41], the virtual system call interface [21], asynchronous signal delivery [42], and SHM IPC [7]. Researchers also pointed out that the behavior of many programs depends on run-time execution properties such as their address-space layout [52] or the order in which threads observe modifications other threads make to shared state [50]. These execution properties could be viewed as implicit program inputs. Reading input from any of these sources can cause so-called benign divergences, where the variants appear to behave differently despite not being attacked. Thus, to support the widest range of programs, an MVEE must guarantee that variants read identical input from all of these resources. For several of these problems, including SHM IPC, no practical solutions have been proposed.

Special-Purpose Multi-Execution Frameworks. Certain frameworks share commonalities with MVEEs, but do not meet our definition of an MVEE. Detile, for example, runs two diversified browsers in parallel and synchronizes their JavaScript engines by executing JavaScript bytecodes in lockstep [17]. The idea is to cross-check the values produced by JavaScript bytecodes. If a bytecode exploits a vulnerability to leak a variant-specific value such as a memory address, the framework will detect the information leak because the cross-check of the bytecode that leaked the value will fail.

Secure multi-execution frameworks, such as the one presented by Devriese and Piessens [15], can also run multiple browser instances in parallel. The idea here is to assign every instance to a different security level. This security level determines which outputs an instance is allowed to produce, and whether it receives default inputs or regular inputs. The goal of secure multi-execution is to detect flows between high-security inputs and low-security outputs.

Typically, these special-purpose frameworks replicate I/O at the level of library calls rather than at the system call interface. They also synchronize the variants far more loosely than a regular MVEE does. Only a subset of the browsers' internal library interfaces will be subject to cross-checking and replication. Thus, whether the browsers do I/O over SHM IPC or not is irrelevant to their correct functionality. In a regular MVEE, however, I/O handling for regular system calls is fundamentally different than I/O handling for SHM IPC. The latter requires dedicated and efficient I/O replication infrastructure that is not available in any existing security-oriented MVEE, but that we propose in this paper.

3 Threat Model

For the rest of this paper, we assume that an adversary tries to attack the protected application by exploiting a vulnerability in the application. We assume that this adversary either operates from a remote machine, or from a local user account that does not have sufficient privileges to tamper with the MVEE directly. We further assume that the MVEE employs a variant generation strategy that would cause the variants to diverge when processing the exploit payload [14, 41].

We also assume that standard defenses such as Data Execution Prevention and Address Space Layout Randomization are in place, though we do not depend on them. We consider the hardware and the OS part of the trusted computing base (TCB). Thus, attacks that target them, e.g., microarchitectural attacks [19, 26, 30, 43, 48], are considered out of scope. Finally, we assume that we can instrument the code or the binaries of the protected application using compile-time instrumentation or binary rewriting. Our assumptions are consistent with previous work in the area [21, 27, 51].

4 Shared Memory Support

SHM can be mapped and accessed from multiple processes simultaneously. It is either file-backed, where other processes map or access the same file or device (e.g., a video card); or anonymous, where other processes map the same memory segment using a system-wide unique identifier. Both versions can be used for IPC. File-backed SHM can also be used as an efficient way to access files or devices without the overhead of system calls. MVEEs need to handle SHM with care because it allows an application to perform I/O without invoking any system calls, thus bypassing the existing MVEE's monitoring and replication of system call I/O.

4.1 Internal and External Shared Memory

We consider two types of SHM. *Internal* SHM (ISHM) can only be accessed by a single (possibly multi-process) variant and is inaccessible to unmonitored processes. An example is an anonymous memory region mapped via `sys_mmap` and then shared with a child process (using `sys_clone` and `CLONE_VM`). *External* SHM (ESHM) can be accessed by multiple variants and, possibly, unmonitored processes. MVEEs can safely allow variants to use ISHM without monitoring individual memory accesses, as long as all communicating threads/processes synchronize in an equivalent order in all variants [21, 31, 50]. Existing MVEEs can impose an equivalent synchronization order in all variants using deterministic multithreading [35] or record/replay techniques [50]. Supporting ESHM is far more challenging, however, as a single compromised variant could attempt to attack an external unmonitored process, which, if compromised, could give that variant unfettered access to the host system. Such an attack is not possible with ISHM because any attack launched by a compromised variant would be confined to that variant.

Use Case	Required For	External	Lines of Code
`fontconfig`	always required	Yes	36k
`pulseaudio`	specific mode	Yes	149k
`X`	specific mode	Yes	400k
`wayland`	specific mode	Yes	113k
`nginx`	specific features	No	137k
`apache`	always required	Yes	308k
`firefox, chrome`	always required	Yes	16M, 30M

Table 1. Shared memory in examined use cases

4.2 Real-World Applications

We examined some widely used libraries and applications that use SHM on the Linux platform, and looked at exactly how they use it. We observed that many applications use SHM to complement other forms of IPC such as pipes and sockets. SHM IPC is used mainly for large data exchanges (e.g., video data), while system-call based IPC is preferred for security-critical data (e.g., control messages) and for synchronous transfer of small amounts of data.

Furthermore, it is worth noting how applications use and pass around pointers referencing SHM. First, we observed that a pointer to SHM might be produced in one component of a process, such as the main executable, but can ultimately be consumed by a completely different component, such as a third-party library. Second, SHM is accessed with exactly the same types of instructions that access private memory, including all ordinary load and store instructions as well as locked versions thereof. Third, many instructions that consume pointers to SHM also consume pointers to private memory, even within the same execution of a program. These observations imply that we cannot statically determine the set of instructions accessing SHM precisely.

A final observation concerns the approaches used to manage SHM. Typically, such management approaches partition SHM into ranges known as *chunks*. Ownership over individual chunks can then be assigned to any of the processes sharing the memory, and their ownership can be transferred from one process to another. In some implementations of such SHM management that we observed, the process that owns a chunk writes pointers referencing the chunk itself to the SHM as meta-data. The value of such pointers is only valid in the address space of the owning process: Even though other processes can read such pointers, the SHM might be mapped at a different address in their address space. This behavior complicates replication and checking in MVEEs, as it implies that pointers written to SHM can differ between variants when they layouts of their address spaces differ.

Table 1 summarizes our findings. All studied use cases employ SHM for IPC, except for `fontconfig` that uses it only for efficient file access. We studied what specific types of SHM the use case requested and whether the SHM is required or optional for specific features or modes. We also studied whether there was ESHM and whether pointers are written to SHM. We discuss this below in detail. The first three use cases are libraries used in (among others) the `mplayer` media player, while the rest are stand-alone applications.

To render text in a specific font, `fontconfig` locates and selects the required font [3]. In doing so, it uses file-backed SHM to efficiently read from its cache files. These cache files are opened and mapped into memory as read-only, but are writable by other processes.

The `pulseaudio` sound server supports several protocols for client-server communication. The "native" protocol provides zero-copy mode that uses SHM [18]. In this mode, client-server messages use a shared ring buffer, and audio data is exchanged using a SHM pool. To manage the buffer and pool, pointers to the SHM are stored in the SHM itself.

In the `X` and `wayland` display protocols, clients that do not directly render on the hardware can send image data to the server through SHM [12, 24]. Other client-server communication still relies on system calls, e.g., using a socket to notify the display server that the new image data is present in the SHM. `X` also provides the option to send image data over sockets, `wayland` does not.

`mplayer` uses SHM heavily to send video frames to the display server. First, the decompressed frames are copied to the frame buffer in SHM using either the standard library version of `memcpy` or `mplayer`'s own internal implementation. Then the on-screen display (OSD) component overlays the copied frame with, e.g., subtitles. This OSD rendering is done in inline assembly routines that overwrite individual pixels in the frame rather than performing bulk copying.

The `nginx` web server uses SHM to share state between worker processes [1]. This enables features such as rate limiting. The SHM is managed through a slab pool, which stores pointers to SHM in SHM. If supported by the system, `nginx` prefers using anonymous SHM, only accessible to its own processes. This is hence a form of ISHM.

The `apache` web server keeps track of server activity through its scoreboard feature. This scoreboard is shared among the worker processes and threads through SHM. Additionally the HTML files for responses get mapped as file-backed SHM. However, these mappings are never directly accessed; they are only passed as buffers to system calls sending responses back to clients.

Modern web browsers such as `firefox` and `chrome` use a multi-process architecture to enhance security [2, 4]. Their required IPC mostly happens over pipes, but for reasons of performance large amounts of data (such as video) are exchanged via SHM. These browsers assume SHM is available and will not work without it, as using only pipes would lead to unacceptable overhead and poor user experience.

We conclude that ESHM is widely used, and thus holds back the adoption of MVEEs. Even when SHM is not an integral part of an application, it might be used by the libraries the application depends on. In addition, pointers to SHM can be written to and read from SHM. This might also be the case for pointers to private memory, but we did not observe such behavior. We therefore do not need to provide efficient support for this behavior to allow for wider adoption.

4.3 Design Requirements

Based on the requirements of MVEEs to provide security and on the use of SHM in real-world programs as discussed, we identified a set of design requirements for MVEEs to support SHM. We split these requirements between those necessary to provide any support at all, and those to do so securely. Additionally, we describe the requirements for an optional extension allowing applications to write and read pointers to and from SHM.

To support accesses to ESHM, the MVEE has to intercept every such access and handle it as an RVP. We will refer to these RVPs as SHM-RVPs. Regardless of the number of variants, every SHM-RVP needs to lead to a single, *unique SHM access* to the ESHM. The results of this unique access then have to be replicated across variants, making sure that all variants get the same inputs and undergo the same impact.

For security, all writes to ESHM have to be synchronized and checked for equivalence across variants, i.e., all variants have to reach the same SHM-RVP and their attempted accesses have to be compared before the unique SHM access happens. Although reads have to be checked for equivalence as well, they do not require synchronization, because any later communication to the outside world will be synchronized, e.g., at a system call or a write to SHM. By the time all variants reach such RVPs, all reads will have been checked.

MVEE support for SHM should not be limited to specific types of data being stored in SHM, and hence not be limited to data that is identical in all variants. In particular, when every variant has its own diversified address space, equivalent pointers (i.e., to the same data) are variant-specific. If the variants attempt to write pointers to SHM, differing values hence have to be allowed as long as they are equivalent. Conversely, if the variants attempt to read back a pointer from SHM, replication has to result in variant-specific pointers.

A plethora of applications do not write any pointers to SHM, however. Furthermore, support of pointers in SHM decreases performance as we will discuss in Section 5.4. Consequently, support for pointers in SHM is best provided as a configurable option.

4.4 Shared Memory Support in Existing MVEEs

Existing MVEE designs do not meet these requirements. GHUMVEE denied the variants access to read-write ESHM by intervening in system calls used to allocate SHM [52]. This forced programs to fall back to alternative IPC mechanisms such as sockets or pipes. At the time of its publication in 2012, GHUMVEE's solution worked for complex programs, including browsers. Today, this is no longer the case, as we argued in this section. GHUMVEE and its derivatives [49–51] hence do not support important classes of popular applications. Bruschi et al. [7] proposed a possible solution somewhat similar to the catch-all part of the design we propose in the next section. Their idea was to make SHM accesses fault, and to then catch the resulting page faults. Bruschi et al. did not present an implementation, however, as they were, as they stated, still investigating the viability of their solution. To the best of our knowledge, they did not publish a follow-up paper with positive results. In our investigation, we discovered that a solution based only on page faults can be effective (i.e., secure and complete), but not efficient enough to be practical. Besides these two solutions [7, 52], we know of no MVEE publications addressing the issue of SHM.

5 Design

As stated above, an MVEE should intervene in all accesses to ESHM. However, it is not possible to locate such accesses precisely using static techniques. We therefore propose a two-layered solution. The first layer consists of a *SHM handler* implemented in the monitor of an MVEE. At run time, this handler ensures that all accesses to ESHM become RVPs, thus enabling the necessary interventions by the MVEE. One assumption we rely on is that the monitor supports signals such as page faults, i.e., that the monitor can intervene when such signals are raised and delivered. Another fundamental assumption of our design is that the MVEE monitor in which the SHM handler is implemented is of the cross-process type. Most existing security-oriented MVEEs are of this type [5, 7, 14, 20, 27, 31, 32, 41, 49, 51, 52, 54, 55, 57, 58], because of the strong isolation that the OS kernel can provide between processes. From now on, we will refer to the handler in the cross-process monitor as the cross-process handler.

Our cross-process SHM handler can correctly handle all (legacy) code dynamically. It introduces significant overhead, however, due in part to the required context switches between processes. Our solution's second layer mitigates this overhead to the extent possible by rewriting the application code, i.e., by injecting invocations of an *in-process SHM agent* at all program points of which we have indications that they might access SHM. We call those points *known accesses.* Just like the cross-process handler, this in-process agent implements the necessary replication and monitoring, but it does so without context switches.

Still, this agent has significant run-time overhead, even when the rewritten code happens to access private memory at some point in the program's execution. It is hence important to determine the set of known accesses as precisely as possible, as both false negatives and false positives can have a detrimental effect on performance. As static analysis is not sufficiently precise, we rely to a large degree on dynamic analysis instead. Concretely, we rely on the cross-process SHM handler. On a vanilla program or library executed on a range of training inputs under control of the MVEE, the cross-process SHM handler will handle all SHM accesses, so it can also produce a log, thus functioning as a dynamic analysis tool. This log is then used to determine the list of known accesses that need to be instrumented with the in-process SHM handler, after which the code can be rewritten. This rewriting can be done effectively and efficiently in a compiler when source code is available and effectively but somewhat less efficiently in a binary rewriter when source code of third-party or legacy software is not available.

False negative results of the dynamic analysis pose no correctness and security problem, because the cross-process SHM handler of the first layer serves as a fall-back mechanism for the in-process handler of the second layer. Indeed, the cross-process SHM handler will handle all SHM accesses that were not previously known and did not get wrapped. As such, the layers and the code rewriting can also be used iteratively to let the set of known accesses to SHM grow gradually, or to adapt to variations in the inputs over time.

5.1 Catch-All Cross-Process SHM Handler

The first component, the cross-process SHM handler, becomes part of the MVEE's monitor, which interacts with the variant processes and the kernel. Figure 1 visualizes how this handler handles SHM accesses that the leader variant (left) and the follower variant (right) want to perform.[1] Those accesses are indicated with the code idiom `... *p ...` and the markings 1_L and 1_F.

Before the variants can actually perform such accesses, they need to have obtained SHM pages from the OS, e.g., via the mmap() system call. The MVEE monitor intercepts all system calls in all variants, and instead of returning a pointer to a requested SHM page, it returns a variant-specific *non-dereferenceable* pointer to each variant. In addition, the monitor maps the ESHM in its own address space to transparently replicate the externally visible SHM accesses. The variants then propagate the pointers within their program state, and causes a segmentation fault whenever they try to dereference a (derived) non-dereferencable pointer (1_L and 1_F). This ensures that all attempted SHM accesses in the

[1]The figures shows one process per variant for the sake of simplicity. For multi-threaded and multi-process applications, we assume one monitor per thread per process, as in prior work [50]. All those monitors then operate as described in this paper. Other multi-threaded support designs are possible.

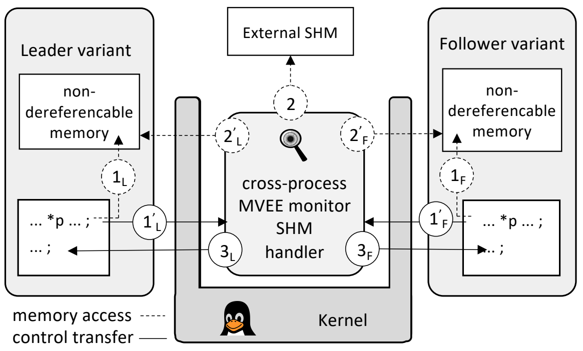


Figure 1. Cross-process catch-all fall-back design

variants become SHM-RVPs, as the monitor is then notified of the segmentation faults by the kernel ($1'_{\text{L}}$ and $1'_{\text{F}}$).

After suspending the process that caused the segmentation fault, the kernel notifies the monitor of the fault. The monitor then checks whether the fault involved a non-dereferencable pointer. If not, the segmentation fault is handled by the MVEE like any other signal [51]. If it does concern a SHM access, the SHM handler performs the necessary replication and monitoring by means of a leader/follower model.

In this model, a follower variant reaching a SHM access is suspended until the leader variant also reaches it. When both variants have done so and the monitor has been notified about that through signals $1'_{\text{L}}$ and $1'_{\text{F}}$, the monitor and its SHM handler first performs the necessary equivalence checks. Specifically, two variants' accesses to ESHM are considered equivalent when both attempt to execute the same instruction on equivalent source operands, which can include non-dereferencable pointers. The handler checks the non-dereferencable pointers for equivalence by mapping them to an equivalent pointer into the ESHM mapping in its own address space, and by requiring this remapped pointer to be identical across all suspended variants. When a divergence is detected the monitor will shut the variants down to safeguard the system from an exploited vulnerability. This implies that SHM accesses occurring in different orders in the variants will be flagged as divergences.[2] Otherwise, i.e., if the equivalence check succeeds, the handler will take the source operand values from the leader variant and perform the intended unique access on the ESHM by executing an equivalent operation (2) to the one that was attempted in the variants. The results from this operation are then replicated to the variants ($2'_{\text{L}}$ and $2'_{\text{F}}$). For variant-specific pointers, this replication includes mapping a pointer from the leader's address space to the equivalent pointer in the follower's address space. Each variant's instruction pointer is then be adjusted such that they resume execution with the next instruction (3_{L} and 3_{F}).

[2]Similarly to how system calls are handled in existing work on multi-threaded programs [50], the orderings of SHM accesses in leader threads are only checked against accesses in corresponding follower threads.

When a leader variant reaches a SHM access first, the handling differs for reads and writes. For a write, the leader is suspended until the followers also reach a SHM access, at which point they are handled as above. Write operations are therefore handled in lockstep: the variant reaching the SHM-RVP first needs to wait for the others. In the case of a read operation, however, the leader does not need to wait for the followers, and can instead be allowed to run ahead as discussed in Section 4.3. The handler in the monitor then performs the intended access (2), propagates the results to the leader ($2'_{\text{L}}$), buffers the result and the information needed for equivalence checking internally, and resumes the leader (3_{L}) before the followers have reached the SHM-RVP. When a follower then reaches the SHM-RVP, the equivalence checking and replication are performed on it using the buffered information. To prevent the leader from running out of control, a configurable limit is imposed on how far the leader is allowed to run ahead.

With the described mechanism, no SHM accesses can escape the monitor's control. However, the fault-based design handling accesses cross-process causes significant overhead. Were a program not running under an MVEE, the ESHM would be accessed by executing a single instruction. Cross-process fault-based handling requires the variant to cause a segmentation fault, execution context to be switched to the monitor that needs to read from and write to the variant's execution context, and finally for the execution context to be switched back to the variant.

5.2 In-Process SHM Handler

If we know upfront that some access will target ESHM, the overhead of the cross-process mechanism discussed above can be avoided by handling the SHM-RVP in-process. Figure 2 visualizes how this works, using pseudo-code instead of native machine code for the sake of clarity. Please note that this in-process handling is active together with the cross-process SHM handler discussed above; known accesses are handled efficiently by our in-process agent, while the cross-process SHM handler works as a catch-all fall-back mechanism for unknown accesses.

First of all, when the MVEE intervenes in the allocation of SHM as discussed above, it also ensures that the requested SHM is mapped into the leader variant's address space, albeit it at a location unknown to the application code of the variant. The implications of this design choice will be discussed extensively later in the security analysis. Furthermore, the monitor initializes in-process handlers (named `SHM_RVP_type[XYZ]` in Figure 2) in the variants and allocates a replication buffer that is mapped into all variants, also at a location unknown to the application code in the variants. There is one in-process handler per type of instruction.

In each variant, each known access is wrapped in a *redirection check*, such as $\boxed{1_{\text{L}}}$ and $\boxed{1_{\text{F}}}$. If the check determines that

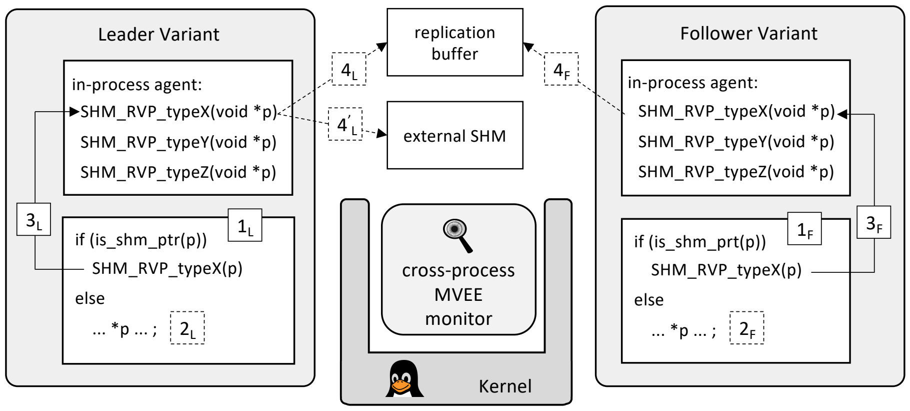


Figure 2. Known access wrapping and in-process agent design

the pointer to be dereferenced is referencable, meaning no access to SHM will take place with it, nothing special needs to be done, the application code can then perform the private memory access itself ($\boxed{2_L}$ and $\boxed{2_F}$). Otherwise the checks redirect execution ($\boxed{3_L}$ and $\boxed{3_F}$) to the appropriate handler of the in-process agent. The invoked handlers in the two variants then together perform the rendezvous, performing the necessary checks and replication in a similar fashion as the cross-process SHM handler. For doing so, the variants' handlers communicate via the replication buffer ($\boxed{4_L}$ and $\boxed{4_F}$). Furthermore, the leader variant's handler also performs the unique access ($\boxed{4'_L}$ to the ESHM. The order in which the checking and replication activities are performed depends on whether read and/or write operations to SHM take place.

5.3 Injecting Redirection Checks

As already discussed in Section 4.2, it is not possible to determine the precise set of instructions that might ever access SHM in a program. Conservatively inserting redirection checks before every single memory access for which one cannot prove that it will never access SHM would introduce an unnecessarily large and possibly unacceptable performance overhead. Being that conservative is not required, however, because the catch-all fall-back mechanism guarantees proper handling of all SHM accesses that are not handled by a redirection check and an in-process handler. We therefore only add redirection checks to code that is known to access ESHM, i.e., code in which we have observed actual run-time accesses to SHM in prior runs. To observe those actual run-time accesses to SHM, we do not need a separate tracing tool. Instead it suffices to let the fall-back mechanism log any instructions for which it has to intervene in a SHM access profile. Based on the collected profile information, individual instructions can then be wrapped in redirection checks. We investigated two methods for wrapping individual SHM accesses.

In the first method, a profile-guided compiler pass injects redirection checks and handler invocations during a recompilation of the code. This injection is done into the compiler's IR. To map the information on individual machine code instructions in the collected profiles to operations in the IR, we rely on debug information. Because the injection is performed before the code is lowered to machine code in the compiler back end, the result in the final binary is injected code that is seamlessly integrated in the surrounding code, which leads to minimal performance overhead. This recompilation is only feasible when source code is available, however, which excludes inline assembly in source code, legacy code, and third-party components such as binary-only libraries.

Our second method for wrapping individual SHM accesses relies on binary rewriting for the necessary injection into inline assembly and legacy and third-party binaries. Binary rewriting typically introduces somewhat more overhead, however, because the resulting code is less optimized.

We reiterate from the intro of Section 5 that security is not affected if SHM accessing instructions are missed during the profiling: those false negatives of this dynamic analysis are still handled securely by the catch-all fall-back mechanism of the cross-process handler, albeit slower.

In addition to the two methods for wrapping individual accesses to SHM, we also wrap burst accesses. In Section 4.2, we already noted that many applications use SHM for large data exchanges. They typically do so using standard library APIs such as `memcpy()` and `memmove()`. For those C library functions, we provide alternative, optimized implementations that replicate and check bursts of SHM accesses. Redirecting entire functions allows us to combine long series of SHM-RVPs into a single SHM-RVP with only one redirection check per pointer parameter in the functions' signatures, which has proven crucial to obtain acceptable performance.

5.4 Supporting Pointers in Shared Memory

When structurally asymmetrical variants feature different address space layouts, as is common for security-oriented MVEEs, and when pointers are written to SHM, which is not uncommon as discussed in Section 4.2, replication and equivalence checking become more complex. The data accessed in

SHM can then comprise variant-specific pointers, so checking for byte equivalence will result in benign program behavior being handled as a malicious divergence. Additional pointer equivalence checks are hence needed.

In our design, when variant-specific pointers are written to SHM by all variants, and they pass the pointer equivalence check, the value from the leader variant will be written in the unique access to the ESHM. In our observations of real-world applications, external processes never altered such variant-specific pointers. However, the pointers are read back from SHM by the application code in all variants, at which point correct replication and checking is again needed.

The MVEE monitor thereto allocates a per-variant shadow copy of the ESHM (not shown in the figures). The cross-process and in-process handlers ensure that each variant stores its variant-specific pointers as well as any other data written to SHM pages in its shadow copy. Whenever data is later read back from ESHM, the leader's cross-process or in-process handler compare the content read from the ESHM to the value in the leaders shadow copy. When the data is equivalent, meaning that the data to be read was written by the application itself (i.e., not altered by an external process), the handlers for the follower variants simply read the corresponding data from their own shadow copy instead of reading it from the replication buffer. This way, the followers operate on their own variant-specific pointers. As the shadow copies need to be accessible to both in-process and cross-process handlers, each of them is mapped as shared between the MVEE monitor and one variant.

6 Proof-of-Concept Implementation

We implemented a cross-process SHM handler in ReMon's CP-MON, a state-of-the-art MVEE [51]. The SHM handler is designed to be extensible. We implemented instruction handlers for all x86 instructions we encountered during the development of this work and have built up infrastructure, under the form of macros, for further extension along the way. Currently, 107 instructions can be emulated, decoded from 43 opcodes. Simple instructions such as `mov r64, m64` or `movzx r64, m8` have fairly straightforward handlers. They analyze the `modrm` to determine the register operand and to decode the memory operand to a monitor equivalent operand, and reserve a spot in the monitor's replication buffer before performing the operation for the leader or replicating its result in the followers. Instructions affecting flags, such as `add/cmp r64, m64`, additionally require replicating the results in the flags register.. Supporting these kinds of instructions can be done in as little as 10-30 LOC using the predefined macros. Instructions specifically used for synchronization, like `cmpxchg m64, r64`, require some more care and custom code to correctly perform replication while still appearing to be executed atomically to outside processes.

We integrated the in-process agent into `glibc` 2.31. The standard C library `glibc` is a convenient location for functionality that has to be available to any loaded binary; it also contains other components such as the synchronization replication agent [50]. In addition, we inserted wrappers for 7 `glibc` functions that cause bursts of SHM accesses: `memcpy`, `memmove`, `memset`, `memchr`, `memcmp`, `strcmp`, and `strlen`. The entire in-process agent has 1021 LOC.

To differentiate between pointers to private and SHM, the in-process agent checks whether the pointer is dereferencable or not. To replicate synchronization on ISHM (e.g., anonymous SHM), our system therefore treats it like ESHM, and makes pointers to it non-dereferencable.

We implemented the compiler pass discussed in Section 5.3 to insert redirection checks in `LLVM` 10. The in-process agent provides a handler for every type of `LLVM` instruction, except for 6 `atomicrmw` operations we did not encounter. Memory accesses wider than 8 bytes are not supported yet, and instructions attempting those are thus not wrapped. This is not a fundamental limitation of our approach but just a matter of additional engineering effort. The `LLVM` pass measures 222 LOC. The complementary binary rewriting tool was developed with Dyninst [38]. This tool is sufficiently advanced to support rewriting the MMX and SSE based inline assembly in `mplayer` into invocations of the in-process agent. Adjacent instructions that access adjacent memory locations are merged into a single invocation.[3]

7 Evaluation

We evaluated both the performance and the security of our prototype implementation.

7.1 Performance Evaluation

We ran all experiments on a machine with a 6-core AMD Ryzen 5 5600x 3.7GHz CPU and 16 GB of RAM. We disabled turbo-boost and hyper-threading. The machine runs Ubuntu 18.04 LTS with Linux kernel 5.3.18. We applied an existing kernel patch to enable the MVEE's in-process system call monitoring, for which we configured the MVEE with its most performant policy [51]. Finally, our MVEE is always configured to run two variants, unless stated otherwise. The variants are diversified by means of disjoint code layouts (DCL) [49], so only one static version of each binary or library is needed, because the MVEE enforces the DCL dynamically.

Benchmarks were compiled with LLVM 10.0 with and without our instruction wrapping pass, using standard release build configurations, except that we ensured debug information was generated to support our profile-guided compiler pass. For that pass, we profiled the applications with sufficient inputs to identify all SHM accesses that also occur in the inputs used for our measurements.

[3]When the paper is accepted for publication, all implementations of all mentioned tools will be open-sourced and published on GitHub.

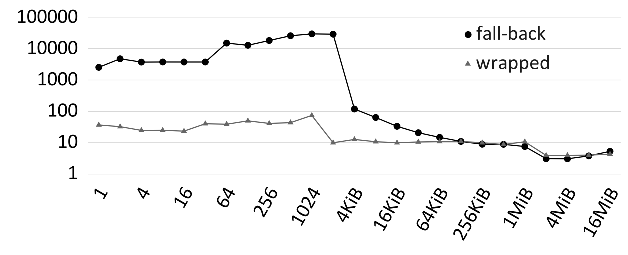


Figure 3. memcpy microbenchmark results

We measured benchmark execution times for (i) the **native** binaries running outside the MVEE; (ii) native binaries running under the MVEE, which then relies entirely on the catch-all **fall-back** mechanism to handle SHM accesses; and (iii) recompiled, **wrapped** binaries running under the MVEE, in which case the in-process handling of SHM accesses is performed by wrapped instructions.

7.1.1 Microbenchmarks. SHM is the go-to choice to transfer large amounts of data to other processes. The wrapper functions we provide for popular `glibc` functions can improve the performance of these transfers. We assess the impact of the replication and checking in the cross-process SHM handler and in-process agent by measuring the execution time of a microbenchmark that uses `memcpy` to copy a buffer into SHM 100000 times in a tight loop. We perform measurements for a range of buffer sizes starting at 1 byte and going up to 4 MiB, taking the average run time of 5 runs for each size.

We compare our two mechanisms, fall-back and wrapped, and show the results in Figure 3. The overhead is relative to the native execution of `memcpy` and the standard deviation never exceeds 1.6% for any buffer size. The figure clearly shows that the in-process agent provides a significant performance improvement for smaller buffer sizes. The relative overhead of the cross-process handler is two orders of magnitude higher than the in-process agent for small buffer sizes, but gradually decreases with buffers larger than 2 KiB. At this point `memcpy` performs its operation by executing a single instruction, thus requiring only a single trap into the cross-process handler. As buffer sizes increase even further, the results for our two mechanisms converge because the memory subsystem itself becomes the bottleneck.

7.1.2 MPlayer Benchmarks. We used `mplayer` 1.4 as a stress test to evaluate the performance impact on software that makes heavy use of SHM to pass large amounts of data. As described in Section 4.2, the decoded frames are copied to SHM and then overlaid with an OSD that includes subtitles. We ran two different experiments. In the first one, we provide as input video files with different encodings and resolutions, and `mplayer` runs in its benchmarking mode with sound disabled. This configuration renders and displays every frame as fast as possible, thus providing an upper bound on the maximum frame rate that can be achieved for each given encoding and resolution. The second experiment plays the video with sound, a realistic usage scenario, and focuses on the dropped frames. Here we opted to use mp4 video files, as one of the most widespread formats, at 1080p. We measured the dropped frames of 10 seconds length, 1080p videos running in 30, 60, 90 and 120 frames, respectively. To evaluate the overhead of the OSD, we measured with subtitles disabled and enabled. In the latter case, we made sure that subtitles are overlaid on every frame, which puts an extreme load on the OSD.

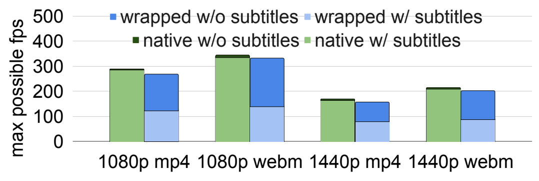
By default, `mplayer` uses its own (inline assembly) implementation of `memcpy` for copying the decoded frames. Individually wrapping every SHM access in that implementation makes it unusably slow. Alternatively, `mplayer` can be configured to use libc's `memcpy` using the `--disable-fastmemcpy` flag. In our tests we observed that building `mplayer` this way caused no noticeable difference in performance when executed without an MVEE. However, when running the software under the MVEE, this change allows for the entire `memcpy` burst of accesses to get redirected through our in-process agent in a single function call.

For the `fontconfig` and `pulseaudio` libraries on which `mplayer` depends, we compiled versions 2.13.1 and 14.2, respectively. We employed our LLVM compiler pass and the Dyninst-based rewriter to wrap the individual accesses, including the inline assembly ones in the OSD rendering.

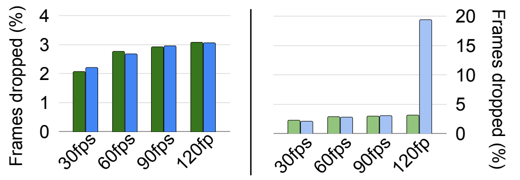
First, we observed that relying entirely on the cross-process fall-back mechanism does not yield acceptable performance with `mplayer`, as the video decoding rate drops below 10% below the rate required for real-time playback. We do not report precise numbers for that version, not to blur the more interesting results obtained with the other versions.

Figure 4(a) shows the maximum frame rate for 1080p and 1440p mp4 and webm videos, for 60 fps inputs. When not displaying any subtitles, we reported frame rates of 125 fps for our worst test-case and 265 fps for our best test-case. We can see that the drop of the maximum frame rate that can be achieved when running under the MVEE compared to native execution is in the range of 25%. While enabling subtitles this increases to approximately 64%, we can still achieve a maximum frame rate of 64 fps for our worst test case and 111 fps for our best test case.

Figure 4(b) summarizes frame drop rates for 1080p mp4 video, computed as (# input frames - # dropped frames) / # input frames. The results on the left show that `mplayer` runs in the MVEE with close to native execution performance when not displaying subtitles. Between 2.2–3.05% of frames are dropped, but all of those are dropped in the first second of the video. This is entirely due to the slow initialization of `fontconfig`. It hence does not impact the user experience. By contrast, the results on the right show that even with SHM access wrapping and in-process handling of SHM accesses,



(a) Maximal achieved frame rate for different formats



(b) Frames dropped for a given frame rate

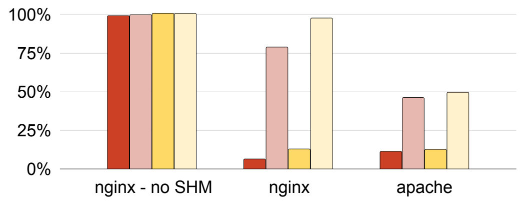
Figure 4. Results for mplayer

the OSD can slow down rendering significantly. As explained in Section 4.2 the implementation of the OSD writes pixel data directly to SHM, all needing replication and checking. Up to 90 fps, a user does not observe any difference between `mplayer` running natively and under the MVEE, as it can handle all operations in real time. However, between 90 and 120 fps this threshold is crossed and we can observe 19.4% of the frames being dropped.

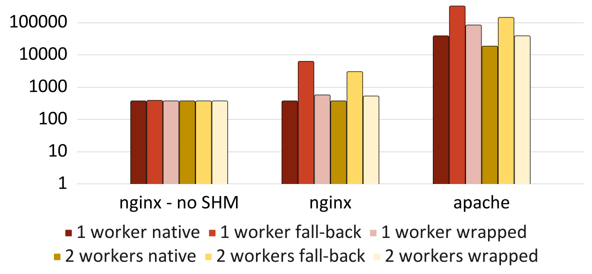
We conclude that running `mplayer` under the MVEE has a large performance impact, in line with the impact observed in literature for non-trivial applications. For the commonly used video formats, sizes, and frame rates that we evaluated, however, there is no observable user experience impact.

7.1.3 Web Server Benchmarks. We evaluated our solution on two web servers, `nginx` 1.18.0 and `apache` 2.4.46. `apache` uses SHM for its scoreboard, a data structure worker processes use to report their status to the main server process. `nginx` uses SHM for rate limiting (which we set to a value far greater than the ones we observed in our experiments). For `apache` we compiled the Apache Portable Runtime (APR) and APR-util libraries—versions 1.7.0 and 1.6.1, respectively—with our custom compiler, as we encountered five instructions in APR that access SHM.

No source code changes were needed that relate to the use of SHM. However, to make these multi-threaded/multi-process applications MVEE-compatible, i.e., to avoid benign divergences related to synchronization, we ran the atomicize compiler pass we developed in prior work and we annotated the types of several variables in the source code to provide information to the pass [50]. Additionally, we manually fixed some data races. For `nginx` we changed 4 lines, marking 4 variables as not being used in synchronization operations. For `apache`, we added variable annotations to 45 lines, and altered 3 lines to make process-wide caches thread-local, thereby fixing data races.



(a) Relative throughput [%]



(b) Latency [μs]

Figure 5. Web server benchmark results

We ran the `wrk` benchmarking client on a separate machine connected to our server through a gigabit Ethernet link. Our client continuously requests a 4 KB web page for 10 seconds over 10 concurrent connections. We ran this benchmark 5 times and report the average overhead.

First, we report the throughput of the fall-back and wrapped versions of the web servers running under the MVEE, relative to their native versions not running under the MVEE. We do so for one and two worker threads. Figure 5 clearly illustrates the benefit of using the in-process agent over the fall-back solution. With only the fall-back solution, only ~5% and ~11% of the native throughputs are achieved for `nginx`. With the in-process handling, however, `nginx`'s relative throughput rises to ~70% for one, and ~98% for two working threads. Additionally we ran experiments with `nginx` configured without the options that make it use SHM. These results show that the extra checks we add using our compiler pass are not the source of overhead for our solution.

We observe a similar trend for `apache`. However, even with the in-process handler, only ~50% of the native throughput is achieved. This performance loss is not due to SHM handling, but due to the fact that `apache` uses the `mmap` system call far more frequently than `nginx`. This security-sensitive system call is always handled by the slow, ptrace-based cross-process monitor. To verify this causal relation, we ran a single unwrapped variant of `apache` under the MVEE and disabled both the checking and replication of SHM I/O. This led to only a ~7% increase in throughput compared to running two diversified wrapped variants under the MVEE, meaning that the majority of performance degradation indeed stems from the slow handling of system calls.

We also observe a discrepancy between the throughputs of the web servers when run under the MVEE. Whereas `nginx` achieves much better relative throughput with two workers, `apache`'s relative throughput barely changes when we add an additional worker. This happens because `nginx`, when run natively, can saturate the network connection even with one worker thread. Adding an additional worker therefore does not improve native throughput, but it does improve throughput under the MVEE. `apache`, by contrast, cannot saturate the network connection even when run with two native workers. This explains why the relative throughput under the MVEE is roughly the same for one or two workers.

7.1.4 Computational Benchmarks.

In the past, MVEE developers used synthetic benchmark suites like SPEC to evaluate the performance overhead of their MVEEs. However, since no SPEC benchmarks use SHM, our solution has a negligible impact on their performance. Our solution adds no more than one additional parameter check to every executed memory mapping system call. The resulting overhead is nothing but noise in the measured execution times. We hence refer to the literature because our MVEE achieves exactly the same performance on those benchmarks [51].

7.2 Browsers

As shown in Table 1, browsers are much more complex than the benchmarks we evaluated. They also rely heavily on advanced OS (e.g., namespaces and ioctls) and hardware features (e.g., GPU acceleration). These features are unsupported by ReMon, the base MVEE on top of which we implemented our prototype. Importantly, these unsupported features are unrelated to how browsers use SHM, and do not pose any fundamental problems to our approach. Adding support to ReMon, or any other MVEE, for these features requires non-trivial engineering. This is the reason that no security-oriented MVEE to the best of our knowledge supports modern browsers. For example, for namespaces we would need to overhaul the bookkeeping facilities that keep track of variants' open files, sockets, memory mappings, etc. We do support commonly-used ioctls such as those for tty terminals, but browsers use plenty of other—often poorly documented—ioctls. As such, we cannot collect measurements on browsers. However, we did study their use of SHM in detail, as described in Section 4.2, and are confident our approach is compatible with their use of SHM.

7.3 Security Evaluation

To assess the effectiveness of our approach against SHM-based exploits, we constructed four vulnerable programs that can be expoited through SHM. While these programs are small examples, their vulnerabilities and corresponding exploits are representative for real-world vulnerabilities in programs that communicate with adverserial programs through SHM. All examples consist of two programs: one attacking program that runs natively and that produces data in SHM, and one vulnerable program that consumes that data and that we will protect with the MVEE. As with the performance benchmarks, we compiled all examples with LLVM 10.0 and tested with and without our instruction wrapping pass to verify both parts of our solution.

Listing 1 lists the first example's vulnerable code fragment. It is based on a proftpd privilege escalation vulnerability (CVE-2006-6563). Through the data in SHM, the attacking program can cause a buffer overflow on line 6, thus setting the `user_id` variable that is consumed on line 8 and passed as a parameter to `setuid`. Notice that the `encrypt` and `decrypt` calls on lines 2 and 8 implement Data Space Randomization (DSR) [6] to protect `user_id`. DSR is a probabilistic protection that encrypts values stored in memory with a random secret key. However, DSR can be bypassed if secret keys can be guess through memory disclosure vulnerabilities or known plaintext attacks [39]. We mounted such an attack on this program by simply including the ciphertext version of the intended user id in the data that the attacker process writes to SHM to overflow the `message` buffer.

When combined with an MVEE, however, the attack is blocked completely. To that extent, we adopted DSR as a diversification technique in the MVEE. The DSR-enabled MVEE then ensures that different encryption keys are used in the variants of the vulnerable program. The MVEE does not stop the buffer overflow: in both variants `user_id` can still be overwritten by an attacker. However, our SHM approach ensures that it is overwritten with the same ciphertext value. When that same ciphertext value gets decrypted with different keys in the different variants, different plaintext values are obtained and fed to `setuid`. As expected, our MVEE then detects the system call argument divergence and terminates the program.

```
1 void example(struct shm_t* shm_ptr) {
2   uint64_t user_id = encrypt(determine_user_id());
3   char message[MESSAGE_SIZE];
4
5   for (size_t i = 0; i < shm_ptr->size; i++)
6     message[i] = shm_ptr->message[i];
7
8   setuid((uid_t)decrypt(user_id));
9 }
```

Listing 1. Example vulnerability 1

The second example, shown in Listing 2, is similar, but instead of using the user id for a system call, the vulnerable program will copy one of two buffers to SHM based on the value of the user id. Again, the MVEE does not prevent the stack buffer overflow into the user id. However, following the injection of the buffer overflow data by the attacker program, when one variant then tries to write the secret data to SHM while the other tries to write the public data, the MVEE detects this divergence and terminates the variants before any data is written to SHM, i.e., before any damage is done.

```c
void example(struct shm_t* shm_ptr) {
  uint64_t user_id = encrypt(determine_user_id());
  char message[MESSAGE_SIZE];

  for (size_t i = 0; i < shm_ptr->size; i++)
  message[i] = shm_ptr->message[i];

  if (!decrypt_id(user_id))
    for (int i = 0; i < secret.size; i++)
      shm_ptr->message[i] = secret.buffer[i];
  else
    for (int i = 0; i < public.size; i++)
      shm_ptr->message[i] = public.buffer[i];
}
```

Listing 2. Example vulnerability 2

An interesting variation on this example can be constructed by omitting the else block. In that case, our MVEE and the monitored programs can hang or terminate, depending on what happens next in the program. The reason is that our implementations of the injected agents, upon being executed in one variant, wait until the corresponding agent is executed in the other variant. So in our example, the variant that tries to write the secret message to SHM will be waiting for a SHM write to happen in the other variant. If the other variant tries to perform such a SHM write in some other function before reaching any other RVP, the diverging writes are detected as expected and the programs are terminated by the MVEE. If, on the other hand, the other variant first tries to execute another RVP, such as executing a system call, that RVP's agent in that variant (or the MVEE monitoring that RVP) will start waiting for the first variant to reach a similar RVP. The result is a deadlock. While this is not the most user-friendly solution, it is at least a secure one, as no secret data gets written to the external SHM. Developing a more user-friendly solution is simple engineering. It suffices to add an MVEE component that periodically checks that variants are not stuck in agents for different forms of RVPs. Such checks, e.g, once per second, will not yield a problematic performance degradation. Alternatively, the injected agents themselves could be extended to detect when RVPs of different types are reached. Complicating frequently executed agents in this way might introduce noticable overhead, however.

```c
void public_resp () { /* respond something public */ }

void private_resp () { /* respond something private */ }

void example(struct shm_t* shm_ptr) {
  shm_ptr->ptr = &public_resp;

  // some time passes

  shm_ptr->ptr();
}
```

Listing 3. Example vulnerability 3

The third example, shown in Listing 3, is based on behavior we observed in `nginx`. The vulnerable program writes a function pointer to SHM and then later calls a function through it. Without MVEE protection, an attacker program can attach to the same SHM segment and alter the pointer to target a function that leaks some private data.

We tested that our MVEE correctly blocks such attacks. First of all, the MVEE ensures that only the leader's function pointer is actually visible to programs that attach to the same SHM segment. When the pointer is not overwritten by an attacking process, the variants use the copy of the pointer in their shadow memory, which allows the program to execute as intended. However, if the pointer is altered by the attacker process, the leader variant detects this (because the value it reads differs from the last written value). The overwritten value is then handled as program input, and it is replicated to all variants. This later causes the follower variant to attempt to jump to a leader code address, which leads to a segmentation fault because of the DCL. The MVEE detects this fault and terminates the program.

The final example, shown in Listing 4, is a simple ROP attack. The vulnerable program copies a buffer out of SHM into a stack buffer which overflows, overwriting the return address and setting up a ROP chain. We assume an attacker can leak enough data from the vulnerable program to construct a ROP attack and inject it via the buffer in SHM if no MVEE protection is used. With MVEE protection and our SHM approach, however, all variants either read the same replicated data values or they read data they wrote themselves. This is enforced at every instruction accessing SHM. Since in this attack the data is altered by another process, all variants are fed the same ROP payload, which contains code pointers that are only valid in one of the variants. Like before, this triggers a segmentation fault that is detected by the MVEE, which terminates the program.

```c
void example(struct shm_t* shm_ptr){
  char buffer[BUFFER_SIZE];

  for (int i = 0; i < shm_ptr->chain_size; i++)
    buffer[i] = shm_ptr->chain[i];

  return;
}
```

Listing 4. Example vulnerability 4

These examples shows that our solution handles SHM RVPs in a way that is analogous to system call RVPs. Ensuring all variants receive the same input, except for known safe cases, fully utilizes available diversifications such as DSR and DCL to enforce and then detect divergences. This, combined with checking all writes to SHM, equips the MVEE with everything necessary to detect malicious usage of SHM.

8 Security Analysis

As discussed in Section 2, an MVEE's security depends on a number of properties: monitor isolation, the diversity of the variants, and RVP enforcement. These properties hold for system-call-based IPC in security-oriented MVEEs. The monitor is isolated and cannot be manipulated by system calls executed by potentially compromised variants; the variants are diversified; and the monitor checks I/O system calls for equivalence and replicates their results. ESHM, however, increases the attack surface: a compromised variant could exfiltrate sensitive data or attack an external (unmonitored) program without using system calls. Once compromised, the external program can perform harmful actions on behalf of the compromised variant, without running the risk of being detected by the monitor. The opposite is also possible. An unmonitored program could attack a variant by sending it an exploit payload through SHM. Such an attack could also lead to a variant getting compromised, though it requires an attacker to already be present on the host system. We compare the security properties of our design for SHM support to regular system call handling in a security-oriented MVEE, focusing on RVP enforcement and monitor isolation.

Similar to system call RVPs, SHM-RVPs are enforced by both the cross-process and the in-process SHM handlers. These read data from SHM exactly once and replicate it to the variants. Even if this data is maliciously crafted by an attacker, all variants receive the same inputs and unintended behavior causes a divergence. Any data written to SHM is checked for equivalence, and only allowed to proceed if this check succeeds. In addition, the non-dereferencable pointers to the SHM differ between variants, as described in Section 5.1. This makes it impossible for an attacker to even inject a non-dereferencable pointer that is valid in every variant. As long as the SHM handlers are used, RVPs are enforced the same way as in MVEEs that only support I/O through system calls.

Strong monitor isolation is provided for the cross-process SHM handlers; if only these handlers are enabled no RVPs can be bypassed. In-process handling is less isolated, however, and provides several targets to an attacker that gained control over one (or all) of the variants: The in-process agent and its replication buffer are present in every variant, and the leader variant even has direct access to the ESHM. An attacker can tamper with the replication buffer to provide different inputs to different variants, while direct access to the SHM allows for unchecked writes.

The degree of isolation and the attack surface of the in-process SHM handlers is similar to that of the in-process system call monitor used in ReMon [51]. We therefore compare these two. Our in-process SHM handler and in-process system call monitor can both be protected against control-flow hijacking attacks by applying software diversity techniques to them. The level of protection we offer against these attacks is hence the same as ReMon. Both components also have their own replication buffers, which could be targeted by data-only attacks. However, in this case too, we offer the same level of protection: both types of replication buffers are mapped at different, non-overlapping addresses in all variants. In order to allow in-process handling of system calls, ReMon's variants need to be able to make unchecked system calls, and this capability is gated through a token which is kept secret. In contrast, our in-process SHM handling requires no secrets. The addresses of SHM pages are not known to an attacker, and these pages are only present in the leader variant. Even if the attacker somehow would be able to determine the address of the ESHM in the leader variant, mounting a successful attack would still be hard. The most straightforward approach would be to have all variants write to this address, in which case the leader would have to perform the required writes to SHM before the followers perform that same access, which for the variants will result in a crash, at which point the monitor will detect an attack. To create a reliable attack, an attacker would need to distinguish between running code in the leader and follower variants, so as to either perform writes to different addresses; or to execute different code paths. However, such variant self-awareness and the ability to have asymmetric effects in the variants is an extremely powerful attacker ability, which would also allow an attacker to bypass other security guarantees in any MVEE.

In conclusion, our design provides the same security properties as other MVEEs. In particular, we enforce the same security properties as ReMon [51]. Although bypassing the in-process SHM handling is possible in theory, it requires attacker capabilities that would allow bypassing other MVEE security guarantees as well.

9 Conclusion

Existing MVEEs do not adequately support programs that use SHM for IPC or efficient file I/O. This blocks the use of MVEEs in many real-world use cases, and is one of the obstacles that stands in the way of their wider adoption.

The presented solution enables full MVEE support for applications that use shared memory, thus removing this obstacle. Our solution consists of a two-tier system to identify, intercept, and replicate accesses to shared memory, as well as different techniques to instrument applications using either compiler passes or binary rewriting. We evaluated the run-time performance of our solution using realistic benchmarks and we carefully discussed its security properties to show that our solution is practical and that it upholds the security guarantees of the MVEE it is integrated into.

Acknowledgments

Part of the research reported in this paper was funded by the Fund for Scientific Research - Flanders (FWO) under grant nr. G033520N. Part of this research is conducted within the Cybersecurity Initiative Flanders funded by VLAIO (Flanders Innovation & Entrepreneurship).

References

[1] [n.d.]. NGINX Development guide. https://nginx.org/en/docs/dev/development_guide.html#shared_memory.

[2] 2008. Multi-process Architecture. https://www.chromium.org/developers/design-documents/multi-process-architecture.

[3] 2018. fontconfig. https://www.freedesktop.org/wiki/Software/fontconfig/.

[4] 2019. Multiprocess Firefox. https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Multiprocess_Firefox.

[5] Emery D Berger and Benjamin G Zorn. 2006. DieHard: probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[6] Sandeep Bhatkar and R. Sekar. 2008. Data Space Randomization. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Paris, France) *(DIMVA '08)*. Springer-Verlag, Berlin, Heidelberg, 1–22. https://doi.org/10.1007/978-3-540-70542-0_1

[7] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. 2007. Diversified process replicæ for defeating memory error exploits. In *IEEE Performance, Computing, and Communications Conference (IPCCC)*.

[8] L Cavallaro. 2007. *Comprehensive Memory Error Protection via Diversity and Taint-Tracking*. Ph.D. Dissertation. PhD dissertation, Universita Degli Studi Di Milano.

[9] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented Programming Without Returns. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[10] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. 2005. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the USENIX Security Symposium*.

[11] Frederick B Cohen. 1993. Operating system protection through program evolution. *Comput. Secur.* 12, 6 (1993), 565–584.

[12] Jonathan Corbet. 1991. MIT-SHM(The MIT Shared Memory Extension). https://www.x.org/releases/current/doc/xextproto/shm.html.

[13] Jonathan Corbet. 2015. Intel Memory Protection Keys. https://lwn.net/Articles/643797/.

[14] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. 2006. N-Variant Systems: A Secretless Framework for Security through Diversity.. In *USENIX Security Symposium*.

[15] Dominique Devriese and Frank Piessens. 2010. Noninterference through secure multi-execution. In *Proceedings of the IEEE Symposium on Security and Privacy*. 109–124.

[16] Stephanie Forrest, Anil Somayaji, and David H Ackley. 1997. Building diverse computer systems. In *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*. IEEE, 67–72.

[17] Robert Gawlik, Philipp Koppe, Benjamin Kollenda, Andre Pawlowski, Behrad Garmany, and Thorsten Holz. 2016. Detile: Fine-grained information leak detection in script engines. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. 322–342.

[18] Victor Gaydov. 2017. PulseAudio under the hood. https://gavv.github.io/articles/pulseaudio-under-the-hood/#protocols-and-networking.

[19] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A remote software-induced fault attack in javascript. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*.

[20] Petr Hosek and Cristian Cadar. 2013. Safe software updates via multi-version execution. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[21] Petr Hosek and Cristian Cadar. 2015. Varan the unbelievable: An efficient n-version execution framework. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[22] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. 2015. Automatic Generation of Data-Oriented Exploits.. In *Proceedings of the USENIX Security Symposium*.

[23] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[24] Kristian Høgsberg. 2012. The Wayland Protocol. https://wayland.freedesktop.org/docs/html/index.html.

[25] Dohyeong Kim, Yonghwi Kwon, William N Sumner, Xiangyu Zhang, and Dongyan Xu. 2015. Dual execution for on the fly fine grained execution comparison. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[26] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the IEEE Symposium on Security and Privacy*.

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

[28] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2016. LDX: Causality inference by lightweight dual execution. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[29] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated software diversity. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[30] Moritz Lipp, M. Schwarz, D. Gruss, Thomas Prescher, W. Haas, A. Fogh, Jann Horn, S. Mangard, P. Kocher, Daniel Genkin, Yuval Yarom, and Michael Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the USENIX Security Symposium*.

[31] Kangjie Lu, Meng Xu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2018. Stopping Memory Disclosures via Diversification and Replicated Execution. *IEEE Transactions on Dependable and Secure Computing (TDSC)* (2018).

[32] Matthew Maurer and David Brumley. 2012. TACHYON: Tandem execution for efficient live patch testing. In *Proceedings of the USENIX Security Symposium*.

[33] Matt Miller. 2019. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. In *BlueHat IL*.

[34] NASA. 2004. *Software Safety Guidebook, NASA Technical Standard*. Technical Report. NASA-GB-8719.13.

[35] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. 2009. Kendo: efficient deterministic multithreading in software. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 97–108.

[36] Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. 2019. MVEDSUa: Higher Availability Dynamic Software Updates via Multi-Version Execution. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[37] Luís Pina, Daniel Grumberg, Anastasios Andronidis, and Cristian Cadar. 2017. A {DSL} Approach to Reconcile Equivalent Divergent Program Executions. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 417–429.

[38] Paradyn Project. [n.d.]. Dyninst: Putting the Performance in High Performance Computing. https://www.dyninst.org.

[39] Prabhu Rajasekaran, Stephen Crane, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2020. CoDaRR: Continuous Data Space Randomization against Data-Only Attacks. In *Proceedings of the 15th ACM*

Asia Conference on Computer and Communications Security (Taipei, Taiwan) *(ASIA CCS '20)*. Association for Computing Machinery, New York, NY, USA, 494505. https://doi.org/10.1145/3320269.3384757

[40] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, et al. 2017. Address Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity.. In *NDSS*.

[41] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. 2009. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*.

[42] Babak Salamat, Todd Jackson, Gregor Wagner, Christian Wimmer, and Michael Franz. 2011. Runtime defense against code injection attacks using replicated execution. *IEEE Transactions on Dependable and Secure Computing* 8, 4 (2011), 588–601.

[43] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *Black Hat USA*.

[44] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[45] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[46] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[47] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. 2012. Memory Errors: The Past, the Present, and the Future. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[48] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[49] Stijn Volckaert, Bart Coppens, and Bjorn De Sutter. 2016. Cloning your gadgets: Complete ROP attack immunity with multi-variant execution. *IEEE Transactions on Dependable and Secure Computing (TDSC)* (2016).

[50] Stijn Volckaert, Bart Coppens, Bjorn De Sutter, Koen De Bosschere, Per Larsen, and Michael Franz. 2017. Taming parallelism in a multi-variant execution environment. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*.

[51] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. 2016. Secure and Efficient Application Monitoring and Replication.. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[52] Stijn Volckaert, Bjorn De Sutter, Tim De Baets, and Koen De Bosschere. 2012. GHUMVEE: efficient, effective, and flexible replication. In *International Symposium on Foundations and Practice of Security (FPS)*.

[53] Alexios Voulimeneas, Dokyung Song, Per Larsen, Michael Franz, and Stijn Volckaert. 2021. dMVX: Secure and Efficient Multi-Variant Execution in a Distributed Setting. In *European Workshop on System Security (EuroSec)*.

[54] Alexios Voulimeneas, Dokyung Song, Fabian Parzefall, Yeoul Na, Per Larsen, Michael Franz, and Stijn Volckaert. 2020. Distributed Heterogeneous N-Variant Execution. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*.

[55] Xiaoguang Wang, SengMing Yeoh, Robert Lyerly, Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. 2020. A Framework for Software Diversification with ISA Heterogeneity. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[56] Xiaoguang Wang, SengMing Yeoh, Pierre Olivier, and Binoy Ravindran. 2020. Secure and Efficient In-Process Monitor (and Library) Protection with Intel MPK. In *European Workshop on System Security (EuroSec)*.

[57] Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. 2017. Bunshin: compositing security mechanisms through diversification. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[58] Sebastian Österlund, Koen Koning, Pierre Olivier, Antonio Barbalace, Herbert Bos, and Cristiano Giuffrida. 2019. kMVX: Detecting Kernel Information Leaks with Multi-variant Execution. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

A Artifact Appendix

A.1 Abstract

This artifact consists of an extension to enable shared memory support in the ReMon MVEE, and the set of benchmarks used to evaluate this extension. The extension itself has been integrated into the source of ReMon.

A.2 Description & Requirements

A.2.1 How to access. Our artifact has been integrated in the open source ReMon repository, available at https://github.com/ReMon-MVEE/ReMon. The artifact can be found in the `eurosys2022-artifact/` directory.

A.2.2 Hardware dependencies. The full evaluation requires two separate systems: One to run the MVEE-protected benchmarks and one to serve as a benchmarking client for the server benchmarks. This second system is not strictly necessary, but running the benchmarking client on the same host as the web server will result in lower relative throughput for protected web servers. We connected these two systems through a private 1 gigabit ethernet link in our experiments.

As a general rule of thumb we advise having one core available for the MVEE monitor and N additional cores for each process the MVEE-protected program would start, with N being the configured number of variants. Thus, to benchmark a simple one-process application, we would advise at least a 4-core CPU. Since our server benchmarks will run with multiple processes, we advise at least a 6-core CPU to comfortably run all benchmarks.

To bootstrap, our system needs at least 8GiB RAM and 10GiB disk space.

A.2.3 Software dependencies. We require the system to either run Ubuntu 18.04 LTS, or to use Docker to run the benchmarks in our provided container.

We used the ReMon patched Linux kernel to enable in-process monitoring to increase ReMon's performance. This kernel patch is not required for any of the functionality, but does impact the performance.

A.2.4 Benchmarks. To benchmark the server applications, we used the `wrk` HTTP benchmarking tool. To benchmark mplayer, we use 1080p mp4 video files. Our original benchmarks used a copyright-protected video which we unfortunately cannot redistribute. However, we provide an alternative set of mp4 and webm files with the same characteristics (resolution and fps), based on video from pixabay.com[4] and music from www.bensound.com[5]. The subtitle file we used for our experiments were already created by ourselves and are available in the same directory as our video files.

[4]https://pixabay.com/videos/aerial-drone-lake-waterfall-rocks-85373/

[5]https://www.bensound.com/royalty-free-music/track/adventure

A.3 Set-up

We automated most of the initial setup in a single `bootstrap.sh` script in `eurosys2022-artifact/`; follow `README.md` to perform this setup. As per those instructions, you can choose to execute the bootstrap script natively (if you run Ubuntu 18.04 LTS); otherwise follow the instructions to set up the docker container we provide. It should complete after about 40 minutes.

Next, build all benchmarks with `benchmarks/scripts/build_all.sh` in `eurosys2022-artifact/`. This takes about 15 minutes.

We disabled hyper-threading and turbo-boost for our benchmarks for better reproducibility.

If you are running Ubuntu 18.04 LTS or 20.04 LTS, you can unlock ReMon's full potential by applying a small kernel patch to enable IP-MON (the in-process monitor component of ReMon), as described in `README.md`.

A.4 Evaluation workflow

A.4.1 Major Claims.

- (C1): Our solution provides shared memory support at an acceptable overhead, shown by evaluation on microbenchmarks, server benchmarks, as well as a media application. This is proven by experiments E1, E2, E3, and E4. The result of experiment E1 (on microbenchmarks) is illustrated in Figure 3, the result of experiments E2 and E3 (on mplayer) are illustrated in Figure 4, and the result of experiment E4 (on servers) is illustrated in Figure 5.
- (C2): When using our solution to protect the mplayer video player, there is no observable impact on user experience when playing video files up to 90fps at 1080p. This is shown by experiments E2 and E3, which are are illustrated in Figure 4.
- (C3): In the paper revision plan that we submitted to the paper's shepherd, we pointed out that we plan to extend the security evaluation with 4 security microbenchmarks to discuss the efficacy of our solution. In the revision, we will hence also extend the current artifact appendix to include a description of how to run those microbenchmarks.

A.4.2 Experiments. For convenience of evaluation, a results_interpreter.ods file is provided in `eurosys2022-artifact/benchmarks/` to more easily interpret the results. Only the green cells contain measurement data, the others are computed from them. The different sheets are named according to the experiments. Additionally, our boostrapping should take care of most of the preparation. To further avoid mistakes in setting up, we suggest using the `eurosys2022-artifact/benchmarks/scripts/run.sh` script, as this provides several options to perform what little required setup remains. This script is used as follows: `./run.sh <options> - <benchmark> <version>`.

Experiment (E1): Microbenchmark; *takes about 5 human-minutes and 25 compute-minutes.* Run three configurations of the `memcpy` microbenchmark: the native baseline; under Remon without wrapping burst accesses in libc; and under ReMon while wrapping burst accesses in libc.

To run natively, pass `-native` as *options* to `run.sh`. Passing `-unwrapped-bursts` and `-wrapped-bursts` will set up the libc version ReMon will load to have unwrapped burst accesses and wrapped burst accesses, respectively.

To run the microbenchmark, pass `microbenchmark` as the *benchmark* parameter, you can omit *version*. To reproduce our original experiment, perform the benchmark 5 times for each configuration.

The microbenchmark will output a list with the time it took, in nanoseconds, to copy a given amount of data, in bytes, into shared memory. The times for the five runs of each size and configuration can be averaged and inserted in the relevant cells in results_interpreter.ods.

Experiment (E2): Mplayer max fps; *takes about 10 minutes, human-minutes and compute-minutes combined.* Run two configurations of mplayer, one natively and one under ReMon. Do so on four different input files: 1080p webm, 1080p mp4, 1440p webm, and 1440p mp4. All 60 fps. Perform this twice, once without a subtitle file loaded, and once with.

While the build script automatically compiles mplayer and wrapped versions of its dependencies, we did the evaluation on a version of mplayer that has been rewritten using dyninst, based on an initial profiling run of mplayer. Because it is based on a profiling run, this step is not part of the initial setup. Run `mplayer_build.sh` with option `-dyninst`. This will require some dynamic analysis first. The script will open an mplayer instance that will play a video slowly. Let at least a few frames with subtitles render before closing the window. The correct version will then be built automatically.

For both mplayer-based experiments (E2 and E3), we compare a native execution of mplayer with the dyninst-rewritten version. These can be run with `run_mplayer.sh`. For the native version, pass `-native` as *options*. Where for `run.sh` you would pass the benchmark and version, now instead pass, in order, the *version* (`default` or `dyninst`), the path to the video to be played, and optionally the path to the subtitles.

Finally, to perform the measurements for experiment E2, add `-maxfps` as one of the *options* to enable the measurement of the max fps.

When the video has finished playing, take the total time it took to play the video (look for `BENCHMARKs` in the output and take the total time at the end of the line) and average that over five runs. The amount of frames in the video (framerate * video length in seconds) divided by the average total time calculated is the maximum framerate that could be achieved. You can simply fill in the average total time in the results_interpreter.ods.

Experiment (E3): Mplayer frame drops; *takes about 10 minutes, human-minutes and compute-minutes combined.* Run two configurations of mplayer, one natively and one under ReMon, while varying the frame rate of the input mp4/webm file. For a direct comparison with the paper, use the webm files. Perform this twice, once without a subtitle file loaded and once with, using the procedure described for experiment E2, but replacing the `-maxfps` argument with `-framedrop`.

This configuration will print an additional `BENCHMARKn` line. The value of *disp* on this line represents the amount of displayed frames. Average this over five runs and insert it in the relevant cells in results_interpreter.ods.

Experiment (E4): Web servers; *takes about 15 minutes, human-minutes and compute-minutes combined.* Run three configurations of our web servers: natively; under ReMon without shared memory instructions wrapped; and under ReMon with shared memory instructions wrapped. Perform this twice, once with one and once with two worker threads configured on the servers. Do this for nginx and apache.

To switch between one and two workers, you will have to alter the config files. For nginx you will need to change `worker_processes` in `nginx/conf/nginx.conf` and for apache change `ServerLimit` in `apache/docs/conf/httpd.conf`, both in `eurosys2022-artifact/benchmarks`.

To run the servers using the `run.sh` script, pass `nginx` or `apache` as *benchmark*. The aforementioned configurations are represented by the *version* `base`, `default`, and `wrapped`, respectively. Provide the *option* `-native` for `base` and `-ipmon` otherwise. We suggest using wrk configured with one thread, 10 connections, and requesting continuously for 10 seconds. Other benchmarking clients might yield different results.

Take the average latency and requests/second reported by `wrk`, additionally averaging these results over five runs for each configuration, and fill them in in the relevant cells in result_interpreter.ods.