Divide and Conquer: Introducing Partial Multi-Variant Execution

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

Adriaan Jacobs
DistriNet, KU Leuven
Ghent, Belgium
adriaan.jacobs@kuleuven.be

Alexios Voulimeneas
CYS, TU Delft
Delft, The Netherlands
A.Voulimeneas@tudelft.nl

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

2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P) | 979-8-3315-9493-0/25/$31.00 ©2025 IEEE | DOI: 10.1109/EuroSP63326.2025.00064

***Abstract*—After several decades of defensive research against the exploitation of memory errors, a wide range of techniques has been proposed, yet no silver bullet has been found. Multi-Variant eXecution (MVX) is one promising proposal for defending against a wide range of known and potentially unknown attacks. MVX systems run multiple program variants in parallel on the same inputs while monitoring their behavior and deduplicating their outputs. By constructing these program variants using automated software diversity techniques, we can ensure that the variants behave identically under normal operating conditions but diverge when attacked. The MVX system detects these divergences and reacts appropriately.**

State-of-the-art MVX systems have several fundamental problems that inhibit their real-world adoption. First, they often require full source code availability to construct variants and eliminate non-deterministic program behavior. Second, they incur significant resource overhead that linearly increases with the number of variants running in parallel.

We propose Partial Multi-Variant eXecution (PMVX), a technique that can mitigate these problems by limiting the scope of MVX to certain well-delineated parts of a target application and by running the rest of the application in Single-Variant eXecution (SVX) mode. PMVX relaxes the source code availability requirement of traditional MVX systems and yields substantially reduced resource consumption while maintaining the strong security guarantees of these systems. However, PMVX implementations must address the non-trivial problem of ensuring all variants are in equivalent states whenever they switch from SVX to MVX mode.

We designed and implemented a proof-of-concept PMVX system called `FORTDIVIDE` that solves this state-equivalency problem using state migration and resynchronization. We thoroughly evaluated the security and performance of our system as a whole, and of our state migration and synchronization mechanisms in isolation. We conclude that PMVX has great potential but needs to be applied with the utmost care since the added overhead of state resynchronization can quickly outweigh the benefits of running in SVX mode.

1. Introduction

In a world increasingly reliant on software, C and C++ remain omnipresent in our software stacks. Low-level systems software such as operating system kernels, web servers, and browsers largely favor C/C++ over other languages due to its unparalleled performance and features. Unfortunately, these come at a hefty price. Loose language specifications and a lack of built-in safety checks, the very basis for its performance advantages, riddle extensive code bases with undefined behavior and memory errors [1]. These memory errors are continuously exploited to take control of systems or exfiltrate sensitive data [2–9], and have spawned a veritable arms race between hackers and defenders for the past several decades [10].

From early on in this arms race, software diversity showed to be a promising mitigation strategy [11–13]. At its core, this strategy introduces some randomness in the software to generate different variants. While these variants adhere to the software's original semantics, they can differ in implementation aspects that attackers rely on to exploit memory errors. Address Space Layout Randomization (ASLR), which randomizes the base address of memory mappings, is one commonly known diversification technique. Others include randomizing register allocation [13], instruction selection [14], or data representation [15]. Unfortunately, the reliance on randomness alone does not *guarantee* protection against exploits, which can still be tailored to compromise one or few variants, possibly aided by information leakage attacks [16].

Multi-Variant eXecution (MVX) systems further reinforce software diversity by running multiple variants in parallel on identical inputs while monitoring their behavior and deduplicating their outputs [17–34]. These systems generate structurally asymmetric program variants, which are highly likely to behave differently when exposed to the same exploit payload. Such divergences in the execution can then be detected by a monitor, which can take appropriate action [35]. A careful choice of software diversity techniques can *deterministically* inhibit a wide range of attack vectors [17, 23], making MVX systems a promising general defense against memory exploits.

Unfortunately, MVX has seen little to no adoption in the real world. One reason is that existing systems often require non-trivial source- or compiler IR-level transformations throughout the *entire* protected program. For example, the system might produce program variants using compile-time software diversity techniques [13], or it might require the instrumentation or removal of program code that triggers so-called benign divergences (i.e., divergent behavior that is not the result of an attack) [21, 36]. For these reasons, programs whose source code is not fully available might be incompatible with MVX. Secondly, MVX significantly increases system resource consumption due to the variant replication mechanism that underpins their deterministic security guarantees. Even ignoring the MVX system's (limited) per-variant monitoring and cross-

DOI 10.1109/EuroSP63326.2025.00064

variant synchronization overhead [24], the increase in memory consumption and computing power required to run programs under MVX protection is directly proportional to the number of variants run concurrently. With just two variants, programs require double the number of free CPU cores to avoid creating scheduling bottlenecks that devastate the system's run-time performance. This currently makes MVX ill-fit for applications that effectively utilize the underlying platform's available parallelism or for systems that are otherwise resource-constrained.

At its core, the issue in both cases above is that current state-of-the-art MVX systems indiscriminately replicate all program state across all variants, subjecting the program in its entirety to the compatibility and computing resource requirements of full multi-variant execution. However, not all parts of the program tend to be equally security-sensitive [37, 38], or equally in need of full replication and protection. For instance, parts of the program that directly manipulate user input are generally at a much higher risk of exploitation [39–41], and benefit more from the monitoring and diversification features of the MVX system. Recent CVEs for `nginx`, for example, show that most security vulnerabilities are in the MP4 and HTTP/3 modules[1], making isolating specific modules that are under active development an interesting option as well. By contrast, parts that have been thoroughly audited or formally verified are at much lower risk. Existing programs already recognize such asymmetry in the exploitability of their components. For example, modern browsers explicitly isolate JIT engines and media codecs from the rest of the browser through the process boundary. More generally, *compartmentalization* mechanisms divide a program into at least two domains, commonly labeled trusted and untrusted, and define explicit call gates that regulate access control between the two [42–45]. In such a design, the trusted part is assumed to be free of exploitable bugs and allowed to access all memory. In contrast, the untrusted part receives significantly fewer privileges and has its access to the trusted part revoked.

In this paper, we adapt the security guarantees of MVX systems to fit the asymmetric security requirements of modern software components, curbing MVX's excessive resource consumption and full source code requirements. We propose Partial Multi-Variant eXecution (PMVX), which limits MVX to the most security-critical or least trusted parts of the application, leaving the rest running under Single-Variant Execution (SVX). Under PMVX, *trusted* portions of the application do not incur the full replication overhead, nor are they subjected to the MVX compatibility or source code requirements, while the MVX system's security guarantees remain maximally upheld in less trusted parts through strong monitoring and full replication. In summary, we make the following contributions:

- We explore the design space of PMVX systems and identify state migration as the most fundamental design challenge.
- We implemented one concrete design in a prototype called `FORTDIVIDE`[2], which we thoroughly evaluated using microbenchmarks and server benchmarks. Appendix A outlines the availability of the `FORTDIVIDE` source code and experiments.
- We conclude that PMVX has great potential but needs to be applied with care since frequent state migration can severely degrade the system's runtime performance.

2. Background

Redundant execution has seen several use cases over the years, such as improving application reliability [28, 46] and dynamic software updating [47]. Security-oriented MVX systems harness ideas from redundant execution to defend against the exploitation of memory vulnerabilities [24]. They use *software diversity* to generate diversified variants of a given program and run them in parallel, replicating all input and deduplicating all output. They suspend the variants at specific points during execution, called *Rendez-Vous Points (RVPs)*, to compare the state of the variants and verify their equivalence before allowing them to proceed. The amount of introduced diversity and the granularity of the RVPs determine the strength of the security guarantees an MVX system can enforce.

Diversification. MVX systems operate on semantically equivalent variants of the same program that vary in their implementation details. The idea is that benign program semantics are generally not dependent on low-level properties, such as the location of code pages in the address space. However, exploits typically only work for a specific memory layout or code structure. Given identical inputs, this causes the variants to reliably and visibly *diverge* during exploitation, as not all the variants react the same way to the malicious payload.

Prior work has applied various software diversity techniques to the MVX context over the years. In their seminal work, Cox et al. introduced non-overlapping address spaces through a compile-time transformation [17], which deterministically inhibited both code-reuse and data-only attacks relying on absolute addresses. Several other MVX systems have since furthered the usability and security of MVX by diversifying the address space at run time instead [24, 26], and introducing additional diversity such as reversed stack layouts [20], randomized heap layouts [18], and more [20, 22, 24, 26, 32–34, 48].

Rendez-Vous Points. To reliably detect divergences, security-oriented MVX systems execute the variants in *lockstep*, suspending and synchronizing them at RVPs to verify the equivalence of their state. In case of a divergence, the monitor takes action to prevent the attack. Otherwise, the variants are allowed to continue executing until the next RVP. RVPs can be applied at different granularities, such as on every call to a sensitive standard C function, but the state of the art for security-oriented MVX systems is to use system calls as RVPs [21, 32, 48, 49], and to let the monitor compare system call numbers and arguments for equivalence. System calls are a natural RVP choice since they are executed infrequently enough to

1. CVE-2024-7347, CVE-2024-32760, CVE-2024-31079, CVE-2024-35200, CVE-2024-34161, CVE-2024-24989, CVE-2024-24990, CVE-2022-41741, CVE-2022-41742, and CVE-2018-16845

2. https://github.com/ReMon-MVEE/ReMon/tree/fortdivide

permit low-overhead monitoring. However, they are still the primary way for applications to interact with the environment. Any successful exploit eventually has to execute system calls to damage the victim, which the MVX system will check for divergences. Additionally, MVX systems must replicate program input and deduplicate program output, most of which happens through I/O system calls. To facilitate input replication and deduplication, MVX systems elect one variant as the *leader*, which actually gets to perform deduplicated system calls, e.g., creating or modifying files, while the other *follower* variants are only used to verify equivalence.

Despite the advantages of syscall RVPs, they can incur a significant performance penalty for syscall-intensive applications like web servers [24]. Under full lockstepping, all variants have to wait for each other at every system call before their states can be compared and the system call can be executed. To mitigate this overhead, state-of-the-art MVX systems do not enforce strict variant synchronization on all system calls [22, 24, 28], but instead implement *relaxed* RVPs. Relaxed RVPs allow the leader variant to continue execution without synchronously waiting for the other variants to reach the same RVP. Instead, the leader records its state when entering the RVP, and lets follower variants asynchronously verify equivalence once they reach the same RVP later on. This mechanism greatly improves performance, and only comes at a minor security cost as the delayed detection window is typically small. Even so, MVX systems typically still enforce total variant synchronization on some highly security-sensitive system calls, e.g., `mprotect` or `execve`.

Monitor Design. Many designs for MVX monitors have been introduced in prior work, which attempt to strike a balance between performance overhead and security guarantees. Cox et al. presented the original MVX design [17], which featured a kernel-space monitor. In-kernel monitoring grants strong monitor isolation and RVP enforcement, with low syscall interception overhead. However, it also embeds the large and complicated MVX monitor in kernel space, where any bugs in it affect the security and reliability of the entire system [24], not just the monitored application alone.

More pragmatic designs place the monitor in user space and try to match the advantages of kernel-space monitors otherwise. They generally rely on the kernel's debugging infrastructure for system call interception, e.g., using Linux' `ptrace` API to monitor the program's system calls from a different process [20, 23, 50]. Unfortunately, while comprehensive, Linux' `ptrace` infrastructure incurs a significant performance overhead, largely due to the many context switches required to intercept even a single system call [24, 28, 51].

As a result, MVX systems have spawned substantial innovation in in-process syscall monitoring mechanisms that avoid `ptrace`'s context-switching overhead. For one, some designs attempt to invoke an in-process monitor by specially handling common syscall invocation sites, e.g., preloading libc wrapper functions [52, 53] or statically rewriting `syscall` instructions to invoke the monitor directly [28]. However, these designs are unsuitable for security-oriented MVX where there is a risk of compromised variants introducing or discovering raw `syscall` instructions and using them to perform syscalls directly, bypassing the monitor. Pure in-process monitors additionally struggle to comprehensively isolate themselves from tampering attempts by compromised variants [28].

Hence, MVX systems with strong security guarantees use the kernel to authoritatively inform the monitor of all syscalls made by the variants, and employ countermeasures to prevent any malicious tampering with in-process monitoring components. MvArmor uses Dune [54] to run the variants as hardware-virtualized processes with the MVX monitor as a hypervisor [22], which comprehensively isolates the monitor from the variants and reliably intercepts all system calls. However, it struggles to support all operating system features, e.g., multi-threading, primarily as a limitation of the prototype, but also as a symptom of the engineering-heavy maintenance necessary to support process-level virtualized implementations of ever-evolving OS features. In contrast, ReMon [24] runs all variants and monitors in regular user-space processes and uses a hybrid syscall interception mechanism that enables simultaneously efficient and secure monitoring. A Cross-Process Monitor (CP-MON) utilizes `ptrace` for strict RVP enforcement, while a more efficient In-Process Monitor (IP-MON) solely handles relaxed RVPs. A small kernel patch brokers syscalls between CP-MON and IP-MON appropriately based on a user-configurable relaxation policy. IP-MON further employs additional countermeasures to prevent confused deputy attacks or memory corruption attacks by compromised variants [24]. We implement our PMVX prototype, `FORTDIVIDE`, on top of ReMon due to its maturity and continued maintenance.

Benign divergences. Variants running in an MVX system must receive identical inputs to avoid divergence. Most MVX monitors achieve this by replicating the results of input system calls executed by the leader variant to the follower variants that did not execute those calls. However, previous research has repeatedly shown that variants can also receive inputs from sources other than input system calls. Examples include the CPU time stamp counter and random number generator [20], the virtual system call interface [28], asynchronous signal delivery [55], and inter-process communication through shared memory [19, 56]. Additionally, variants may use run-time and variant-specific execution properties as implicit inputs. For instance, memory addresses can differ across variants due to the software diversity techniques used to construct them [21, 57], or variants may observe shared state changes in different orders in multi-threaded environments [36]. Variants that operate on such implicit inputs can diverge benignly, even if they are not under attack. MVX systems typically address these benign divergences by neutralizing sources of implicit inputs through additional RVPs, implemented via compiler instrumentation [56] or run-time interception [20, 21]. Still, even state-of-the-art MVX systems cannot reliably prevent all benign divergences, particularly those caused by differences in the memory layout of the variants [21, 24].

3. Threat Model

We adopt the same threat model as earlier work on MVX [22, 24]. We assume an adversary tries to exploit memory errors in an application to launch an attack. This adversary can either interact with the application from

a remote machine or from an unprivileged local user unable to directly tamper with processes running under MVX control other than through the interface exposed by the program. On the defensive side, we assume the MVX system has correctly diversified its variants such that the adversary's exploitation attempts trigger detectable divergences [17, 20, 23].

We further assume that standard defenses, such as Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR), are in place, even though these are not required for our system's security. We assume that our MVX implementation, the kernel, and the hardware we run on, is trusted and free from exploitable bugs. Finally, we consider micro-architectural attacks to be out of scope [58–61].

4. Design

Our proposed PMVX system operates much like a traditional MVX system in that the end user launches the protected application by starting the MVX monitor. The monitor then spawns one process for each program variant and subsequently interposes itself between these newly created processes and the OS kernel. From this privileged position, the MVX monitor can intercept, examine, and, if necessary, manipulate all interactions between the variant processes and the operating system. This enables the monitor to replicate input, detect divergent behavior, deduplicate outputs, and stop compromised variants before they can harm the host system. Throughout this process, the MVX monitor ensure that all uncompromised variants remain in equivalent states [20].

The primary difference with a traditional MVX system is that, in our PMVX system, protected application variants are compartmentalized into a trusted/SVX and an untrusted/MVX compartment. Our PMVX monitor allows only the designated leader variant to execute so long as the application execution remains in the trusted/SVX compartment. If the execution crosses the compartment boundary, the variants notify the PMVX monitor, which then either resumes previously suspended follower variants, creates new follower variants (when switching to MVX mode), or suspends/terminates existing follower variants (when switching to SVX mode). The fundamental challenge here is twofold. First, we need to ensure that these transitions incur low overhead. Of course, the overall overhead of compartmentalized applications is primarily determined by the granularity of the compartment boundary and the frequency of inter-compartment switches [62], the determination of which is an orthogonal and active area of research [63]. Still, we explore different design directions to make transitions between SVX and MVX mode as efficient as possible (cfr. Section 4.1).

Second, we must ensure that all variants are brought into equivalent states whenever we switch to MVX mode. This involves propagating program state from the sole variant executed in SVX mode to all variants we resume/create when we switch to MVX mode. If any single-variant program state changes are incorrectly replicated to follower variants during untrusted execution, they will likely cause the follower variants to diverge from the leader variant and each other, which would constitute an unrecoverable benign divergence.

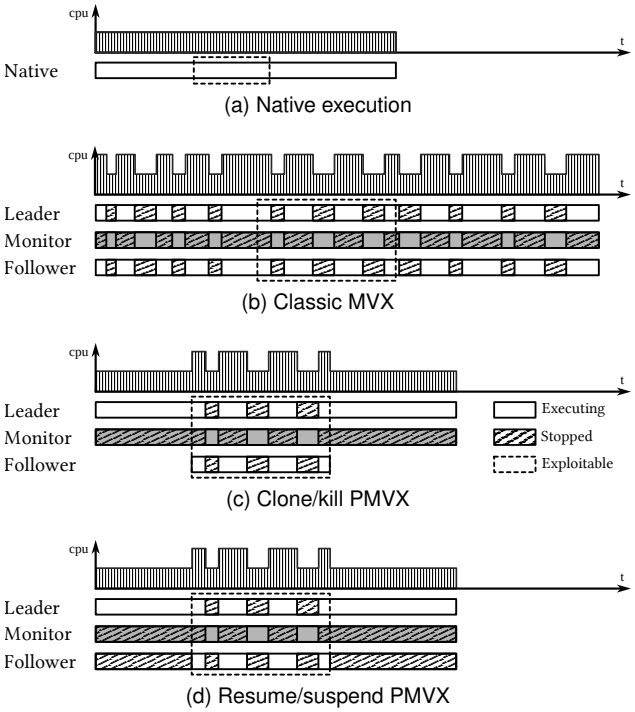


Figure 1. Conceptual representation of process timelines and CPU load when running a program natively (a), under classical MVX (b), and under two possible implementations of PMVX (c, d). Notice how PMVX decreases CPU load compared to full MVX.

4.1. Efficient Mode Transitions

We distinguish two main approaches to control the execution of follower variants and implement the transition between SVX and MVX mode, which we refer to as *clone/kill* and *resume/suspend*. Fig. 1 visualizes the difference between both.

clone/kill. When the program is about to enter MVX mode, a simple approach is to `clone` the existing leader variant (i.e., the variant we executed in SVX mode) to create the required number of follower variants (cfr. Fig. 1c). After the clone, the PMVX system immediately operates in full MVX mode and monitors the variants for divergences while executing untrusted code. When the program transitions back to trusted code, the PMVX system can kill the followers and switch back to SVX without multi-variant monitoring.

While simple, the clone/kill approach has a few drawbacks. First, cloning a process is expensive, even on modern operating systems. It requires allocating a new process control block with a copy of the leader's page tables and other duplicated state. Cloning the leader for each follower variant during every transition to MVX mode causes a substantial delay in MVX activation, potentially making the performance drawbacks of using PMVX outweigh its resource efficiency gains. Second, since every follower is an exact clone of the leader, they lack the diversity that underpins the exploit detection capabilities of MVX. To maintain variant diversity, all followers should be re-diversified on the fly, which adds even more overhead to the already costly clone-style MVX transition. Online variant re-diversification also adds significant complexity to a PMVX system and requires the implementation of efficient diversity generation mechanisms directly on the

critical path [52, 64], which may create a new trade-off between the amount of diversity and the speed of diversification where there previously existed none [13].

resume/suspend. Alternatively, we can launch the PMVX system in MVX mode and immediately spawn diversified follower variants, as per regular MVX startup. Whenever the program transitions into trusted code, the PMVX system switches to SVX by simply keeping the followers suspended and having the leader variant continue execution (cfr. Fig. 1d). This approach allows us to fully exploit existing variant diversity, unlocking any diversification technique for use with PMVX. On any later transition into MVX, we simply resume the followers.

Although this method keeps the suspended follower processes alive during SVX, it does not increase system utilization. The followers do not consume any CPU time, and their resident memory can be swapped out on-demand to satisfy potential surges in memory consumption during SVX. In addition, resuming and suspending processes is faster than cloning and killing them, potentially making transitions between SVX and MVX more efficient.

Nevertheless, the resume/suspend approach also comes with downsides. When resumed, the follower variants will not be aware of any program state changes made during SVX mode, which they likely depend on to ensure correct behavior during MVX. To avoid divergences between the leader and the followers, the leader variant's updated state must be *migrated* to the followers when entering MVX mode, which presents its own engineering and design challenges. Still, we expect this state migration to be *at most* as complex and costly as the variant re-diversification necessary for clone/kill while avoiding any restrictions on diversity techniques and without incurring the performance penalty of repeatedly cloning and destroying new processes. As a result, we opt to implement the resume/suspend approach in our PMVX prototype, `FORTDIVIDE`, and outline our approach to state migration in Section 4.2.

PMVX-program interface. PMVX-protected applications must notify our monitor to instigate a switch between MVX and SVX. We do this by injecting special RVPs into the protected program through recompilation, hooking, binary rewriting, or any other available method. Upon reaching SVX-to-MVX RVPs, the leader variant executes a system call. The monitor intercepts this system call and resumes suspended follower variants. Similarly, when the variants reach MVX-to-SVX RVPs, they execute a system call that prompts the monitor to suspend all followers. Note that attackers cannot abuse the latter RVP to disable MVX maliciously. Doing so would require them to successfully exploit a vulnerability in all variants while the system runs in MVX mode. Proper variant construction makes such exploits impossible by design.

Note that we do not exclusively assign specific code sections to either of the two PMVX modes. Code such as `libc` functions is often shared between the trusted and untrusted program parts, and must be callable from MVX and SVX mode. By introducing additional RVPs that signal compartment boundary crossings to the PMVX monitor instead of rigidly assigning code pages to each domain, we naturally support such code-sharing behavior. Fig. 2 shows an example program flow where different code regions execute in MVX or SVX based on the switching RVPs along the execution path. In this example the application only switches back to SVX after the application has finished its own setup. It also shows how the lack of a switching RVP for `utility2` makes it available in SVX and MVX mode. A re-entrant call such as from `dangerous1` to `dangerous2` could be supported by the RVPs, though ideally manual involvement would make this a direct call. Additionally, switching back to SVX early, such as the call to `safety`, can further add to PMVX' benefits.

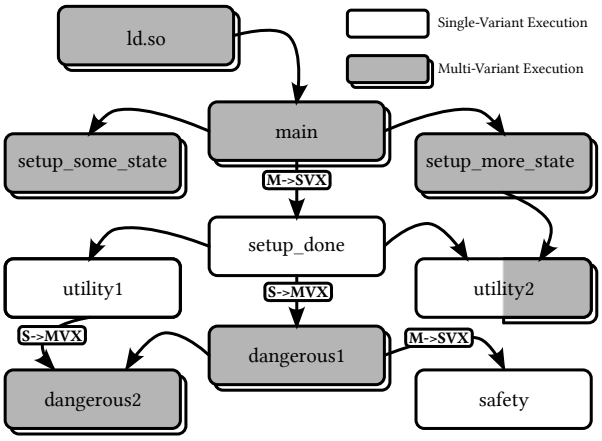


Figure 2. Example call graph with different parts of the program running in SVX or MVX mode and some being able to run as both.

4.2. State Migration

Under resume/suspend, we must migrate the modified leader variant's state to the suspended followers at each transition from SVX to MVX. To this end, we designed an efficient *state migration* mechanism that records relevant state updates in the SVX monitor and applies them to the followers when switching to MVX.

We distinguish two main categories of state the leader can accumulate during SVX: **process-management data** and **program-execution data**. The first encompasses all external state of the leader variant process as managed by the OS through system calls, e.g., the status of file descriptors, memory mappings, threads, signal handlers, etc. The second includes all changes to the leader variant's memory, generally made through regular, unprivileged memory writes in user space.

Any security-oriented MVX system already intercepts all system calls, which naturally includes those that change the process-management state. During SVX, we therefore keep our PMVX monitor attached to the leader variant to record such state changes. To replicate these changes in the followers, we temporarily resume them and force them to execute the same system call that caused the state change by single-stepping them through a `syscall` instruction and carefully setting up the argument registers. After completion, we restore all our modifications to the follower's registers and memory, and suspend them again.

Technically, we *could* employ a similar strategy for program-execution state. The monitor could intercept and replicate every write instruction during SVX that may possibly contribute to the leader's relevant state during the switch back to MVX. However, this would introduce

exorbitant overhead in SVX, which contradicts PMVX's goal of improving efficiency. Instead, we opt for a lazy *fast-forward* approach, in which we try to determine which relevant memory contents have changed since the previous migration and migrate only those contents. We identify four main sources of such program-execution state:

The stack stores data such as local variables and return addresses. Well-behaved programs only access stack data in the current stack frame, or, if given a direct pointer to them, individual stack elements in frames of parent functions. Therefore, when entering MVX, we only migrate the leader's current stack frame and data reachable through pointer arguments to the followers.

The heap stores most objects that applications allocate dynamically. Heap objects tend to be highly interconnected, and the overall heap structure is generally opaque to the PMVX monitor. Furthermore, without hardware-supported pointer tagging/capability mechanisms such as CHERI [65], it is fundamentally impossible to determine the precise set of heap objects that may be accessed from the MVX compartment, whether statically [66] or dynamically [67]. Hence, we elect to migrate the entire heap from the leader to the followers when entering MVX.

Global variables can be accessed from anywhere in the application and could, therefore, always be updated by the SVX part. However, indiscriminately migrating all globals would constitute a large and slow data transfer on every SVX-to-MVX transition. Hence, we opt to migrate globals selectively, i.e., only those that are writable and of which the contents differ in the leader and the followers. We identify equivalent globals among the variants through the available symbol information in the binary.

Other memory-mapped regions can be instantiated using calls such as `mmap`. Just like heap objects, the PMVX monitor has no information about the structure of these mappings, nor can it reliably predict which parts the MVX partition will need. Consequently, we migrate all custom memory-mapped regions when entering MVX.

Migration Mechanism. Across the sources of program-execution state that need migration, we distinguish two different tasks: migrating whole memory regions and migrating *targeted values*, e.g., differing parts of objects. We design a separate state migration technique for each.

To accommodate the migration of entire memory regions, we reserve a portion of the address space in all variants, which we call the Monomorphic Partition. Our system forces the application to map all regions that must be fully migrated during every SVX-to-MVX switch in this Monomorphic Partition. The OS does not need to be aware of this region's existence or location. Instead, the monitor will ensure that only the appropriate memory regions get mapped here, similar to how state-of-the-art monitors already control the memory layout to aid in diversification. When the PMVX system switches from SVX to MVX, we migrate all mappings in this memory region by copying them from the leader to the followers. Copying large amounts of data will lead to significant performance overhead, so the efficiency of this copy could dictate a substantial part of our overhead. We implement an efficient copy-on-write approach called *partial fork*, which we describe further in Section 5.1.

We generally do not migrate regions with executable permissions, as their contents should not change after their initial mapping. Code caches for Just-in-Time compilers are one notable exception. We currently do not support such programs, but with enough engineering, we could support them transparently [68]. This only leaves regions with a combination of readable and writable permissions. Our MVX system does not diversify the layout of regions in the Monomorphic Partition to avoid issues with address-sensitive behavior [21].

This allows us to implement an optimization where we reserve the exact same Monomorphic Partition in all follower variants. With the addresses of equivalent data in all variants being the same, any existing pointers to such data do not require explicit migration. Still, not all data can be migrated by simply mapping it in the Monomorphic Partition. The stack, for example, is mapped at different base addresses in each variant, and so are memory regions holding global variables. This, in combination with the diversity in executable mappings inherent to MVX, litters the leader's program-execution state, including the Monomorphic Partition, with leader-specific pointer values that are not valid in the followers. Such values first have to be translated before they can be used in MVX mode. In addition, those mappings that cannot be migrated through the Monomorphic Partition might still contain data that requires migration, for example global variables.

Both cases, pointer values and regular values, introduce a second, more complex, class of state migration we need to accommodate: migrating targeted values. Instead of migrating an entire memory region, we will only copy specific values in them, as we illustrate in Fig. 3. To this end, we must know their precise locations and sizes in each variant's address space. The size is known for pointers, but for other values can range from a single byte to the width of a memory address to even buffers and data structures of arbitrary size. In Section 5, we study the problem of determining this precise information, but for now, we assume it is known. Using this information, we write the targeted values into an intermediate buffer in the leader, where one part contains regular values (1a) and the other pointer values (1b). We then notify the monitor about the SVX-to-MVX transition (2), at which point the followers can perform the partial fork (3). Meanwhile, the monitor copies the regular values over to the followers as-is (4a) and the pointer values after translating them (4b). After (4a), (4b), and (3), we resume the followers (5). Upon resuming, the followers copy the targeted values from their intermediate buffer to their intended destination (6a)(6b). Notably here, the partial fork (3) and the copying of the targeted values from the leader to the followers (4a)(4b) can be done in parallel.

5. Implementation

We implemented a prototype of `FORTDIVIDE` based on ReMon [24], a state-of-the-art system call-based MVX system for the Linux platform. ReMon uses a cross-process monitor that permanently attaches itself to all variants using the `ptrace` API. It can optionally offload the monitoring of low-risk system calls to an in-process monitor running in user space. We designed our system

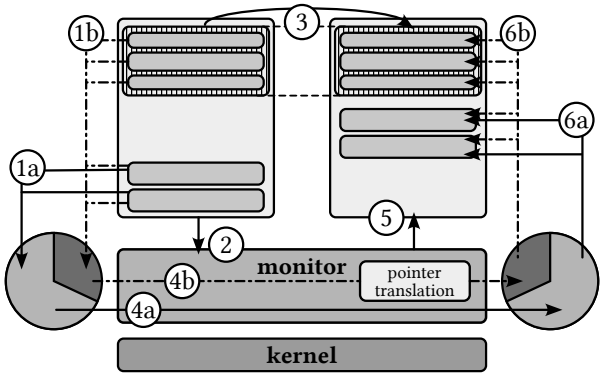


Figure 3. Overview of the order of operations for the migration of state at the SVX-to-MVX RVP.

so programmers can compartmentalize the protected program by inserting call gates whenever the program should switch from SVX to MVX or vice versa. We did not investigate automated compartmentalization and call gate insertion techniques, viewing it as parallel research.

We wrote a small kernel patch that adds two new system calls to support MVX-SVX transitions and state migration. The first system call, `mvx_exit`, suspends the follower variants and ensures our monitor will only receive syscall entry and exit notifications when the leader variant executes system calls that can change process-management state (cfr. Section 4.2). The second system call, `mvx_enter`, performs the partial fork (cfr. Section 4.2), resumes the follower variants, and re-enables notifications for all system calls executed by the variants.

We implemented two types of call gates for switching between MVX and SVX modes, which can be integrated into a program by modifying its source code or hooking function calls. The call gates for switching from MVX to SVX are identical for both leader and followers variants, and simply invoke `mvx_exit`. However, the gates for switching from SVX to MVX differ slightly. The leader's version copies targeted values (cfr. Section 4.2) that need migration into an intermediate buffer and then invokes `mvx_enter`. The monitor then identifies the corresponding call gate locations in the follower variants, moves their program counters to those locations, and resumes their execution to enter the gate. The follower's version of the gate, in contrast, invokes `mvx_enter` directly. During the syscall, the kernel performs the partial fork to copy over the Monomorphic Partition from the leader to the follower, and passes control to the monitor. The monitor translates the targeted values and copies them over to the followers' intermediate buffers before resuming all variants. Finally, the followers copy the targeted values over to their destination addresses, after which all variants transition into the MVX partition.

5.1. State Migration

As mentioned in Section 4.2, `FORTDIVIDE` must migrate two types of state from the leader variant to the followers. Migrating **process-management state** is fairly straightforward, since we can just replay the syscall that caused the state change in all followers. Migrating **program-execution state** is much more complicated. In Section 4.2, we described our state migration as two techniques working in tandem, entire memory region migration and targeted data migration, but left out some specifics.

To support the migration of memory regions, we map the heap and memory-mapped data in the Monomorphic Partition. `FORTDIVIDE` does this by identifying relevant `mmap` calls based on their call arguments and overwriting the preferred base address argument at the system call entrance. The primary heap presents an additional challenge as libc maps it using the `brk` system call by default. Since this system call does not have a preferred base address argument we can overwrite, we modified libc to map the primary heap using `mmap` instead. This required a trivial code change to libc.

Three apparent options are available for migrating program-execution state from the leader to the follower variant processes. The first would be to map the Monomorphic Partition as shared memory between the variants. However, that would open a communication channel between the variants and lead to issues when multiple variants start reading from and writing to the regions simultaneously. Alternatively, we could set up a shared memory channel that is writable in the leader and only readable in the followers as an intermediate, copying the leader's state into it first and then copying it into the Monomorphic Partition in the followers. Here, the memory transfers would become a significant bottleneck, especially as the total amount of to-be-migrated data increases. Finally, we could make the monitor transfer the data using `process_vm_writev`, a system call that moves data from one process to another. This more direct write can eliminate intermediate transfers, but fundamentally, it will do little to alleviate the memory transfer bottleneck. Ideally, we would only copy the data modified since the previous state migration.

As all mappings in the Monomorphic Partition are mapped at the same addresses in all variants, we drew inspiration from the `fork` mechanism. We implemented a *partial fork* in our MVX entrance system call that makes a copy-on-write version of the leader's Monomorphic Partition available to the followers by pointing the relevant page table entries of the followers' Monomorphic Partition directly at the leader's data and resetting the leader's page table entries as copy-on-write. This migration mechanism eliminates all migration overhead in the leader and minimizes the amount of data we effectively have to transfer to the followers. As long as the variants do not write to the Monomorphic Partition, no data must be transferred. We also optimized our partial fork mechanism to make it as efficient as possible by not rewriting page table entries that are still up to date (i.e., they still point to a physical page frame the leader has not written to since the previous migration).

5.1.1. Migrating Targeted Values. We inject a *migration agent* into the variants to support the migration of targeted values. This agent is a shared library written in C. The call gates we insert into the protected application call the agent to copy to-be-migrated values from the leader to the followers through the intermediate buffer we set up.

To copy these values, we pass the agent two lists of (location, size) tuples. One list describes all non-pointer

values, and the other represents all pointer values. The agent migrates values in the first list as-is. For values in the second list, our system performs an additional post-migration translation step that adjusts the migrated pointer to point to the same logical object and offset as the original pointer copied from the leader, even if the follower variants employ a form of address space diversity.

We build the lists as follows. First, we collect the location and sizes of the arguments of the function we are building the call gate for, which must be migrated to ensure the follower variants execute the function with the same arguments. If any of the function arguments point to the stack, we also add the pointed-to data to our lists to ensure the follower variants see the pointed-to data even though we do not map stacks in the Monomorphic Partition and, hence, do not migrate the entire stack upon switching to MVX mode.

Next, we collect the locations and sizes of to-be-migrated global variables. While all variables could technically be migrated, this would result in unnecessary transfers and potential errors with pointer values (e.g., pointers might no longer reference the intended pointee after migration). To address this, we use offline dynamic analysis to identify relevant global variables and their types. We run the target application under `FORTDIVIDE` in a test environment with diverse inputs. Whenever the application switches from MVX to SVX mode, our monitor performs a piecewise comparison of equivalent global variables in the leader and follower variants, identifying mismatches. Mismatches may indicate variant-specific pointers, which we confirm by sliding a window over the region around the mismatching bytes. If the window reveals valid pointers to equivalent memory regions, we add their locations to our second list. We conduct a similar comparison during switches from SVX to MVX mode. In this case, mismatches indicate global variables overwritten since the last switch to MVX mode. We add these locations and sizes to our first list.

Finally, we build lists of targeted values in the heap or `mmap`'ed data regions. If no diversity is applied to the variants, our partial fork mechanism can migrate these regions as-is. However, with code or data layout diversity, these regions may contain variant-specific pointers. To ensure correct migration, we add the locations of such pointers to our second list. Since the heap structure changes throughout the execution of the program, we cannot automatically generate the migration handler code that populates this second list. Instead, we support two options for migration of these values. First, a programmer could write the migration handler manually based on source code analysis. The complexity of such an implementation depends on the scope and type of diversity we apply during variant construction. Since our current system only applies code layout diversity, the complexity is minimal since the only variant-specific values on the heap will be code pointers that are easy to identify. If we also apply data layout diversity, then all pointers to data require migration as well. We leave such an exercise for future work and describe data diversity techniques that do not impose such migration requirements in Section 7.

Second, we implemented an automatic memory scanning mechanism that can run during SVX-to-MVX transitions. Our scanner attempts to identify pointers in the to-be-migrated regions in the leader, and adds them to the second list. Technically, this scanner could replace the manually written migration handlers entirely. However, the scanner is prone to false positives and false negatives (cfr. Section 5.1.2), incurs high run-time overhead (cfr. Section 6.3), and negates most of the MVX security benefits (cfr. Section 7). As such, we use it only as a debugging and development aid, and strongly caution against its use in deployment.

5.1.2. Benign Divergences. Section 2 lays out several reasons why variants running in a regular MVX system can diverge even if they are not under attack. In a PMVX system, variants could also diverge due to incorrect state migration caused by misidentification of targeted values in the aforementioned migration handlers.

False positives occur when values are mistakenly identified as requiring migration or translation when they do not. This can happen if certain code paths explored during offline analysis are never reached in production, such as when features are disabled, or if our analysis overestimates the data needing migration. For example, our analysis might mark entire global data structs as to-be-migrated, even if they only contain one field with a variant-specific value.

The impact is more pronounced for the heap and other mappings in the Monomorphic Partition, as they are migrated indiscriminately. While much of this data may not require migration at all, our partial fork mechanism mitigates most of the overhead compared to directly copying the data.

False positives can also occur during pointer translation. Although rare, this could happen with uninitialized buffers if they initially contain equivalent pointers between the leader and followers at the first SVX entrance but later hold a new valid pointer in the leader upon entering MVX.

Unnecessary data migration and pointer translation add performance overhead during the switch to MVX, but do not cause correctness issues. We analyze the performance cost further in Section 6.1.

False negatives occur when state modified in SVX is accessed in MVX but was missed during migration or translation, leaving it out of sync. This can result in detectable divergences, causing the monitor to terminate the application. Such terminations during deployment can impact availability. However, we can log these instances to expand the migration set and prevent future occurrences.

6. Performance Evaluation

All experiments were run on an Intel Xeon Silver 4210R CPU at 2.40GHz, running Ubuntu 20.04 LTS, equipped with 64GB DDR4 RAM. The kernel is Linux 5.4.212 with both ReMon's patch to support IP-MON and our small kernel patch to support our kernel module, which remained loaded for each test. To make our results reproducible, we disabled hyper-threading and turbo boost, and set the CPU scaling governor to performance. We evaluated our prototype for performance on microbenchmarks and two real-world web servers, as well as for security on a vulnerability we added to nginx.

TABLE 1. TIME TO EXECUTE 10,000 ITERATIONS.

	native (μs)	CP-MON (μs)	IP-MON (μs)
getpid	0.09	17.03	0.64
PMVX switching	0.53	51.03	4.3
fork	46.18	-	-
clone	48	-	-

6.1. Microbenchmarks

To get a clearer picture of the performance impact of the added switching to and from MVX, we set up a microbenchmark that measures how long the switch takes in a few scenarios. We anticipate three dominant overhead sources during the switch we want to investigate: resuming and suspending the followers, our partial fork, and migrating data. Our microbenchmark consists of a single function that consecutively calls the MVX enter and exit system calls in a tight loop for 10,000 iterations. We ran each experiment five times and report the mathematical averages of the execution times here. We evaluated three main configurations. The **native** configuration runs without any monitoring. The **CP-MON** configuration runs the microbenchmark under `FORTDIVIDE`, but handles the system calls using only cross-process monitoring. Finally, the **IP-MON** configuration runs the microbenchmark under `FORTDIVIDE` using the more efficient in-process monitoring infrastructure of ReMon.

First, we try to capture the overhead of resuming and suspending the variants alone. Keeping the Monomorphic Partition empty and migrating no other data ensures we only measure the impact of calling the handling functions in the monitor and our custom kernel module.

We measure an average time of 4.3μs for IP-MON and 51.03μs for CP-MON. To estimate which part of this time is added by our resume/suspend code, we separately measure the raw syscall interception cost of our monitor by calling `getpid` in a tight loop, which nets 0.64μs for IP-MON and 17.03μs for CP-MON on average. Subtracting this cost twice from the total time, once for MVX enter and once for MVX exit, we estimate that our resume/suspend mechanism adds roughly 3μs under IP-MON and 16.97μs under CP-MON, which includes replacing the arguments on each invocation, redirecting the suspended variants, and resuming of the variants.

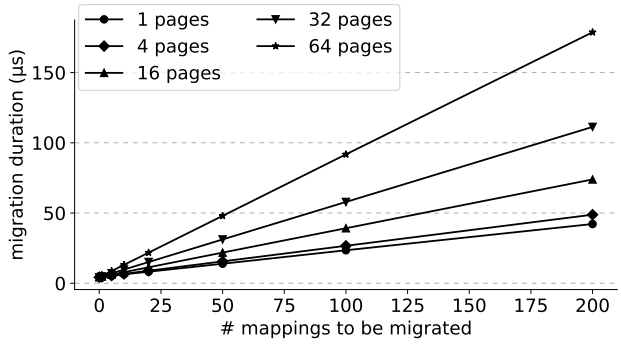
In contrast, we consider the potential overhead of a clone/kill approach by solely calling either `fork` or `clone` in a tight loop under native execution. We measure an average time of 46.18μs for `fork` and 48μs for `clone`. Adding the same per-syscall interception overhead obtained from our `getpid` experiment, we can expect `fork` and `clone` to respectively go up to 47.46μs and 49.28μs under IP-MON and 80.24μs and 82.06μs under CP-MON. This is already significantly more expensive than our resume/suspend approach, without even factoring in the additional overhead of mandatory variant re-diversification.

In our second test, we look at the cost of the partial fork, which is highly dependent on the overall landscape of the Monomorphic Partition. We vary the number of memory regions in the Monomorphic Partition between one and 200, with a size between one and 64 pages, and alternate read/write and read-only permissions to prevent the kernel from merging adjacent regions together. We measure the lower and upper bound for each setting, respectively representing no pages altered and all pages altered in SVX. We simulate the latter by altering our kernel module to always create a new Copy-On-Write (COW) copy for each page. To avoid potential noise from setting up the follower's page table on the first MVX entrance, we perform one entrance/exit cycle as a warm-up. We only report the measurements when running under IP-MON, as the added time difference when using CP-MON always stays around 49.29μs, which is simply the difference in raw syscall interception overhead. Fig. 4a and Fig. 4b show a summary of the results, lower and upper bound respectively.

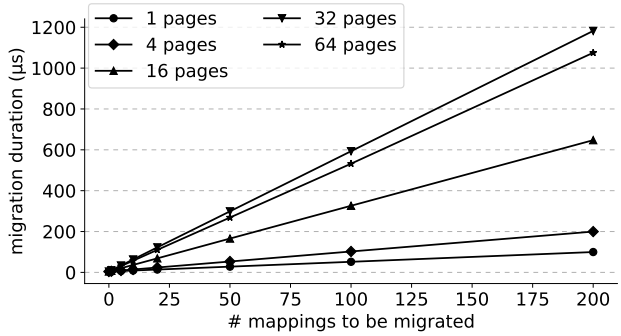
For one mapping, the lower bound time cost increases from 4.56μs for one page to 5.22μs for 64 pages, roughly the same and barely an increase over the base cost of the switch under IP-MON. The worst case in this test, 200 memory regions of 64 pages each totaling 50MiB, takes around 178.69μs for a lower bound. As expected, the lower bound increases linearly with the number of memory regions for a given average region size and increases as this average increases. Further analyzing the impact of the specific distribution of memory regions over the Monomorphic Partition for equal utilization rates, we compare 50 memory regions of 64 pages, 100 memory regions of 32 pages, and 200 memory regions of 16 pages. All these consume 12.5MiB of memory. We measured their migration costs as 48.00μs, 57.88μs, and 73.91μs respectively. We conclude that the cost of the partial fork also increases with the number of memory regions, though not as significantly. This is likely because the kernel restarts its iteration over the page table from the beginning for every memory region, requiring synchronization and TLB manipulations each time. Looking at the upper bound then, we notice a dramatic migration time increase as the number of mappings or their average size goes up. For 200 separate page-sized mappings, our upper bound measurement is already double that of the equivalent lower bound scenario, and the measurements continue to diverge superlinearly as the utilization rate of the Monomorphic Partition increases, e.g., roughly 5x for 200 mappings of 64 pages each.

These results highlight the fact that we only want to map regions that will be relevant for the MVX part into the Monomorphic Partition. Otherwise, we always incur an additional cost, even if those pages were not altered when we switch to MVX, but especially if they are.

Finally, we look at the overhead of migrating data. Since the translation of pointers adds extra steps to data migration, we wanted to evaluate migrating regular values and migrating pointers separately. We write two manual migration handlers that migrate one value in a loop, one as-is and one translated, and vary the number of times we run the loop between 16 and 4096. While the additional overhead of translating pointers is negligible for small amounts of migrated values, it quickly adds up afterward, as is immediately noticeable from Fig. 5. This presents us with the useful insight that qualifies the migration overhead over the type of data we migrate: even if we have to migrate a large memory region, limiting the number of pointers inside this region that require translation can yield significant overhead reductions. Our optimization



(a) Lower bound: no pages altered in SVX.



(b) Upper bound: all pages altered in SVX.

Figure 4. Time cost for partial fork under PMVX in function of the amount of memory regions and relative size of memory regions in the Monomorphic Partition.

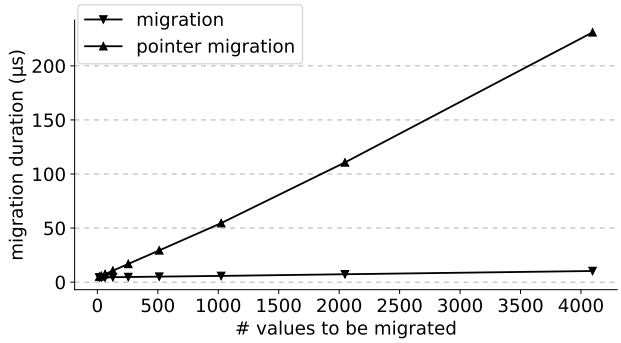


Figure 5. Time cost of targeted value migration for increasing number of 8-byte values.

that maps the Monomorphic Partition at the same location in all variants exploits exactly this insight.

6.2. Server Benchmarks

We also evaluated our prototype in a more realistic scenario by applying PMVX to two commonly used web servers: `nginx` version 1.23.3 and `lighttpd` version 1.4.60. Both are configured to serve the same 4KB static web page and `nginx` is configured with four worker processes, unless stated otherwise. For the server benchmarks we ran `wrk` as a benchmarking client on a separate machine connected to our host via a private gigabit link and configure it to continuously request the page on one thread with 10 connections. Each run lasts 10 seconds and we report the results as the average of five runs.

Naturally, we also need to draw the line between SVX and MVX. We made this decision manually for this evaluation, but based on the assumption that we want to limit MVX to code processing user input. By analyzing the source code we isolated the functions that translate the incoming requests to a structure that the servers can handle, starting MVX just after the request has been received by the server and exiting again just before the reply is constructed. It is important to note that these benchmarks show a worst-case scenario, where the web servers are set up so that the MVX is on the main path handling requests, and thus gets triggered for every request being handled. (Most) web servers are already developed to handle requests efficiently, utilizing as few system calls as possible, making the impact of our relatively expensive switching system calls more pronounced. In a different scenario, we might want to place a library or module that is not triggered for every request in MVX instead.

We run two configurations for each web server. Nginx C1 hooks `ngx_http_process_request_line`, where incoming requests are processed. Unfortunately, this function only returns after the request has been timed out or the request has been fully handled, which we do not want. Thus, we added an additional hook on `ngx_http_handler` in nginx C2, switching back to SVX when it is called, transferring the sending of the response back into SVX. The entrance hook takes only a single heap pointer as an argument, which we can simply copy over to the followers. Lighttpd C1 hooks `connection_handle_read_state` and C2 `http_request_headers_process`. The first only takes a single heap pointer as an argument, whereas the latter takes four pointers as arguments, one being on the stack and the others on the heap.

For the migration of other in-memory state we elected to use manual handling in this evaluation, allowing us to limit handling to only the smallest subset of state required and filtering out as much noise from false positives in the detection of state to be migrated as possible. Later we will look into the effect of other options. Still, we discovered what data required migration by manually analyzing the automated discovery described in Section 5.1 and monitor logs. For Nginx C1 we migrate 19 values, 9,248 bytes combined, and 145 pointers, bringing the total amount of data to 10,408 bytes. Nginx C2 moves the exit back to SVX earlier, allowing us to forego some migrations, we still migrate the same values, but only 95 pointers, bringing the total to 10,008 bytes. Lighttpd C1 migrates four values, 32 bytes combined, and three pointers, only totaling 56 bytes. In contrast, lighttpd C2 migrates slightly more data, seven values for 8240 bytes combined, but no pointers. For the sake of simplicity, these handlers were compiled into the server binaries.

As expected, the earlier exit to SVX in nginx C2, which means no system calls are executed in MVX, outperforms nginx C1. For nginx C2, both the CP-MON configurations even outperform the original ReMon using IP-MON, suggesting that if the division removes enough system calls from being handled in MVX, then the overhead of the switch can be compensated for. Since CP-MON still provides strictly higher security guarantees, we can run MVX with CP-MON protection in our NVX mode, but IP-MON performance, or better, overall. If performance

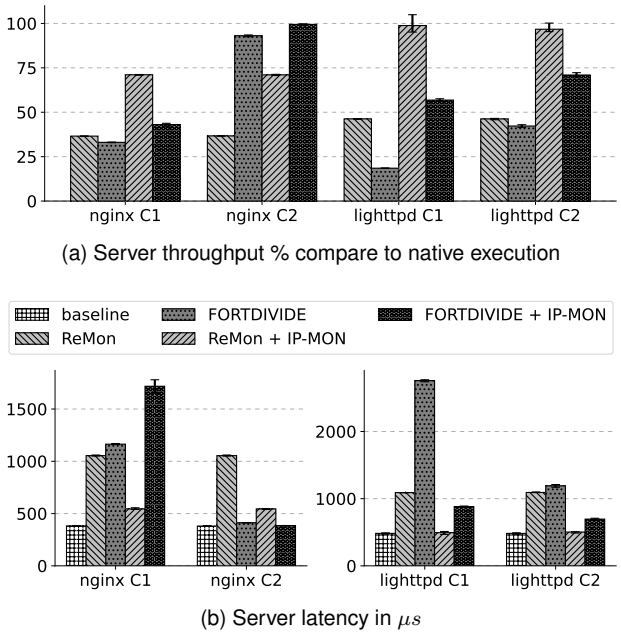


Figure 6. Comparing server benchmarks under ReMon and FORTDIVIDE.

is more of a target, however, than FORTDIVIDE with IP-MON achieves near native performance. In the case of lighttpd, though FORTDIVIDE with IP-MON does outperform ReMon with CP-MON, the frequent switches prove too much for the server's performance. Full results are provided in Fig. 6.

By profiling native nginx and lighttpd using Valgrind's callgrind tool we can estimate the relative number of CPU cycles used by our hooks, which in turn enables us to estimate the CPU cycles we can potentially save using PMVX. Classic MVX would bump CPU usage of the application, ignoring monitoring, to 200% with two variants running. The hook for nginx C1 takes around 82% of CPU cycles, meaning roughly 18% can run as SVX and the CPU usage of nginx C1 is roughly 182%, this is not much of an improvement and explains why our additional switching has such a big impact. Nginx C2 exits back to SVX earlier, the exit hook takes around 74.5% of the CPU cycles, meaning in total only 7.5% of the CPU cycles are ran in MVX and explaining the stark difference between nginx C1 and C2. For lighttpd both configurations are already closer together, with C1 at 19.5% and C2 at 15.5%, meaning the PMVX variants would take 119.5% and 115.5%, respectively.

6.3. Migration Strategy Impact

We have a few options on how to handle the migration of state, which determines how much data is migrated and, thus, the impact on performance. Our nginx C2 configuration in Section 6.2 achieved a throughput of 96.66% compared to the baseline, but we want to see if there is an even better trade-off between manual effort and performance.

First, we used our dynamic analysis to detect migration targets in global data semi-automatically. With the analysis results, we only had to define manual migration handlers for pointers on the heap, leading to 22 copies, 58,772 bytes, and 114 pointer migrations, 73 of which were manually defined. Our automated approach only slightly overestimates which pointers need migration—three more compared to manual handling—but we notice a stark increase in the amount of data copied despite just three more copies, now totaling 59,684 bytes, a 5.97x increase. This can be explained by how we implemented the dynamic analysis. Translating an offset back to a global symbol, where possible, can overestimate the size of some migration targets. In this case, most of the overestimation comes from a single false positive symbol, a 49,152 byte array that is untouched in our MVX mode. Even with this overestimation, we achieve a throughput of 99.50% of baseline for nginx, comparable to manual handling with much less developer effort.

Finally, we evaluated the performance impact of our automated heap scanning mechanism. This mechanism allows us to migrate heap pointers without having to manually write a migration handler, but it should not be used in production due to security concerns (cfr. Section 7). Enabling the scanner results in a staggering 4031 copies, totaling 90,844 bytes of data, and 4050 pointers being migrated, for a grand total of 123,244 bytes. The amount of data copies increases because the pointer scanning approach requires two pointers to be migrated; the variant-specific one, which needs translation, and its location in the Monomorphic Partition, which does not need translation. Note that pointers already pointing into the Monomorphic Partition are ignored by the pointer scan. Evidently, the additional pointers migrated here are never dereferenced in MVX mode, as the previous manual approaches also worked without issue, without migrating this data. The added effect of these translations and the requirement to read and verify every pointer in the Monomorphic Partition causes the throughput for nginx to fall dramatically, to just 2.30%.

Our dynamic analysis encountered some false positives and overestimated the size of some migration targets. However, as shown in Section 6.1, the migration of extra data does not immediately increase overhead. Instead, the increased overhead seems to largely come from the additional pointer scanning we do. We re-ran the previous manual heap pointer migration experiment with additional migration of bogus pointers and values to match the total amount of data migrated in the automated heap scanning experiment and found that the throughput only increased to 23.40%. Thus, the overhead here is not caused by the monitor's handling of the data, but rather by detecting the pointers in the first place. The results are summarized in Table 2.

6.4. Allocator Impact

We have shown with microbenchmarks how not only the size, but also the configuration of the Monomorphic Partition can have an impact on performance. Our monitor changes the allocation behavior of the application by mapping the heap and mmap'ed data regions into the Monomorphic Partition (cfr. Section 5). The custom allocator we built to determine the layout of the Monomorphic Partition was built with two goals in mind: limiting the amount of memory mapped but not actively used and limiting the migration of allocator state. It only has to migrate

TABLE 2. EMPIRICAL MIGRATION STATISTICS FOR DIFFERENT LEVELS OF AUTOMATED MIGRATION ASSISTANCE IN THE NGINX C2 PMVX INSTANCE. WE FIND THAT THE THROUGHPUT IS PRIMARILY AFFECTED BY THE AUTOMATED DISCOVERY OF POINTERS IN THE HEAP THAT NEED TRANSLATION.

	fully manual	auto globals	auto globals + heap
throughput	96.66%	99.50%	2.30%
copies	19	22	4031
pointers	95	114	4050
total migration size (bytes)	10,008	59,684	123,244

TABLE 3. COMPARISON BETWEEN THE DEFAULT ALLOCATOR IN LIBC AND OUR CUSTOM ALLOCATOR FOR SERVER BENCHMARKS RUNNING UNDER FORTDIVIDE WITH IP-MON ENABLED.

	libc allocator		FORTDIVIDE allocator	
throughput	88.03%		96.66%	
	run start	**run end**	**run start**	**run end**
regions	1	2	1	2
total pages	100	415	119	296

(a) nginx results

	libc allocator		FORTDIVIDE allocator	
throughput	56.42%		71.01%	
	run start	**run end**	**run start**	**run end**
regions	1	1	1	1
total pages	134	167	289	339

(b) lighttpd results

two pointers into the Monomorphic Partition, without the need to translate them, in contrast to the manual state migration for libc's allocator, which requires 265 copies, but no pointer migration, totaling 2,240 bytes. Based on our evaluation of state migration, we can estimate that the difference in state migration should hardly impact performance. Instead, what we want to look at is the difference in overhead caused by the layout of the Monomorphic Partition. We look at nginx C2 and lighttpd C2 from Section 6.2, though the configuration has no real impact on the allocation behavior, running under FORTDIVIDE with IP-MON enabled. We run each configuration once with a modified libc to place its allocator in the Monomorphic Partition and once with our custom allocator. In addition to overhead results, we also report the number of memory regions the kernel has to handle and the total number of pages, both at the start, when the server has not processed any requests yet, and after a full benchmarking run.

For nginx we see that our custom allocator improves throughput from 88.03% to 96.66%. Looking at the actual memory footprint used by the allocator we see that libc's allocator has 415 pages mapped while ours has 296 pages mapped. lighttpd's throughput improves from 56.42% to 71.01%, but libc's allocator maps only 167 pages while ours maps 339.

7. Security Analysis

PMVX systems maintain the core security property of regular MVX systems: **exploits will cause a divergence and, therefore, fail if the protected application executes them in MVX mode and if either (i) their payloads must be tailored to variant-specific properties or (ii) they produce variant-specific outputs**. With this property in mind, we now perform a theoretical evaluation in which we describe several representative attacks that abuse memory errors in memory-unsafe applications, examine the conditions under which PMVX systems will stop them, and name (at least) one concrete software diversity technique we can deploy to fullfil those conditions. Afterwards, we also perform an empirical evaluation in which we implemented two exploits. This evaluation confirms that FORTDIVIDE stops the code-reuse attack by default, and can stop the cross-boundary attack by adjusting the compartment boundaries, as predicted in the theoretical evaluation.

7.1. Theoretical Evaluation

Information Leakage Attacks are often the first step toward compromising a victim application [2, 69]. The attacker could, for example, make the application leak the location of relevant code or data, and subsequently use the leaked data to craft a payload for the next stage of the attack. In a PMVX system, information leakage attacks fail if the leaking code executes in MVX mode and if the output contains variant-specific addresses. The PMVX system can achieve the necessary diversity by deploying Address Space Partitioning (ASP) [17, 22]. FORTDIVIDE applies the Disjoint Code Layouts (DCL) transformation [23] to all code regions, but does not apply ASP to data regions in the Monomorphic Partition. As such, FORTDIVIDE allows leaking pointers to data in the Monomorphic Partition, but it will stop all other pointer leaks when it runs in MVX mode.

Code-Reuse Attacks divert the control flow of a victim application to a (set of) attacker-chosen locations such as C library functions or Return-Oriented Programming (ROP) gadgets [8]. Code-reuse attack payloads can contain absolute addresses (e.g., traditional ROP [8] or JOP [70]) or relative addresses (e.g., PIROP [71]). These attacks fail in PMVX systems if the control-flow diversion happens while the application executes in MVX mode, and if the targets include addresses (in case of absolute code-reuse attacks) in or variant-specific offsets (in case of relative code-reuse attacks) to layout-diversified code regions. PMVX systems can defend against absolute code-reuse attacks using ASP [17, 22] or DCL [23]. They can also defend against relative code-reuse attacks by deploying with non-overlapping code offsets [22]. Since FORTDIVIDE applies the DCL transformation [23] to all code regions, it is immune to all absolute attacks performed in MVX mode. It does not defend against relative code-reuse attacks. This is not a fundamental limitation of our system as applying the necessary diversity is simply a matter of additional engineering.

Data-Oriented Programming (DOP) Attacks manipulate the data flow of a victim application without diverting its control flow [7]. In sufficiently complex applications, these attacks can perform arbitrary computations and corrupt arbitrary data structures. DOP attacks fail in PMVX systems if the payload executes in MVX mode and if the attack must either write variant-specific values, or write to variant-specific addresses (in case of

absolute DOP attacks) or write to variant-specific offsets within a data region (in case of relative DOP attacks). PMVX systems can defend against absolute DOP attacks using ASP [17, 22] and against relative DOP attacks by deploying non-overlapping data offsets [22] or heap randomization [18]. Alternatively, they could also deploy Data Space Randomization (DSR) to defend against all types of DOP attacks [72, 73]. `FORTDIVIDE` does not apply any defenses against DOP attacks.

We could implement such defenses in principle, but would have to carefully consider their scope and effects on state migration. ASP, non-overlapping data offsets, and heap randomization would all change the layout of the heap. Thus, by applying these randomization techniques, we would no longer be able to map the heap in the Monomorphic Partition. This would make state migration dramatically slower and require us to implement migration handlers that record and migrate *all* values the leader writes to the heap during SVX.

DSR is a much more interesting alternative since it can encrypt data using variant-specific encryption keys. DSR can also be applied selectively to data that is not accessed outside MVX mode, and thus does not need to be migrated during SVX-to-MVX transitions. Such a selective application would provide some protection against DOP attacks, while remaining compatible with our existing state migration system.

Cross-Boundary Attacks target the interaction between trusted and untrusted code in our current PMVX implementation. Recent work by Mergendahl et al. demonstrated an exploit on multi-privilege polyglot applications [74], where unhardened but trusted Rust code interacts with untrusted, CFI-hardened C code. In their example, an attacker exploits a memory error in untrusted code to corrupt a code pointer, which is invoked only after transitioning to trusted code, bypassing CFI checks. This exploit should succeed even when trusted code is bug-free and untrusted code is hardened.

We evaluate this exploit's feasibility under PMVX. To hijack control flow in trusted (SVX) mode, the attacker must first procure a valid code pointer (e.g., a gadget address) for the leader variant, the only active variant in SVX mode. Since we assume the trusted code is bug-free, the attacker needs a memory error in MVX code to bypass diversification through information leakage. However, any leaked code pointers would differ across variants, triggering detectable divergences in MVX mode and thwarting the exploit. An attacker could instead abuse a memory error in MVX mode to leak a code pointer through an I/O operation that executes in SVX mode. They can then launch an attack by corrupting a code pointer in the leader variant while the application is back in MVX mode. For the exploit to succeed, the corrupted code pointer must be used only after transitioning back into SVX mode. At this point, the leader could be redirected to an attacker-chosen location without detection.

We conclude that, while possible, these cross-boundary attacks require specific conditions, particularly memory errors in untrusted code and limited diversification. Our PMVX implementation currently employs only code diversification. Additional data-level diversification, e.g. heap layout randomization [18] or struct randomization [15], would significantly increase resistance to such attacks. It not only broadens the attack surface coverage, but also provides probabilistic memory safety, reducing the likelihood of memory errors in the first place. The PMVX techniques presented in this paper could leverage further advancements in MVX-based data diversification and detection to further complicate targeted cross-boundary attacks, raising the bar for potential adversaries.

7.2. Empirical Evaluation

Cross-Boundary Attack. We implemented a proof-of-concept setup for the aforementioned attack consisting of a main application that utilizes a vulnerable library, with the library's exposed functions hooked to enter MVX, to verify its effectiveness. In the vulnerable library an attacker can overwrite a code pointer that is later called after `FORTDIVIDE` has switched back to SVX. Indeed, while ReMon blocks this attack early on by detecting the leakage of the pointers, the attack succeeded under `FORTDIVIDE`. We can block the attack by reconsidering the compartment boundary. Concretely, if we do not cross the compartment boundary until after we execute the I/O operation, `FORTDIVIDE` blocks the attack.

Code-Reuse Attack. We further evaluated the security impact of our solution by examining an attack scenario on `nginx`. As discussed earlier, MVX systems such as ReMon, and thus `FORTDIVIDE`, inherit the limitations already present in MVX systems. Consequently, data leaks remain possible unless fine-grained data diversification is applied. For this reason, we disregard mere memory disclosure as an exploit goal for our attacker and instead focus on achieving complete worker takeover and arbitrary code execution.

We assume our attacker has the ability to leak a few pointers and overwrite data at an attacker-chosen address. The attack scenario proceeds as follows. First, the attacker leaks a pointer to a libc function, the buffer the request is stored in, and the location to one of `nginx`' stored function pointers. From the leaked libc function the attacker can construct any arbitrary ROP chain. The leaked request buffer address allows the attacker to know where the payload will be injected. As a launch point for the attack we targeted a hash table that maps specific request header strings to handler functions. By corrupting a single pointer in this table, the attacker can change the handler function and the third argument passed to the handler function. In a subsequent request, the attacker injects the payload through the request body. The attacker overwrites the pointer in the hash table entry to call the first gadget of the chain and pivots the stack to the injected payload to execute the remainder of the ROP chain.

ReMon stops the attack as soon as we attempt to leak any code pointer, thanks to DCL and the checks on the system call sending the response to our client. Under `FORTDIVIDE`, we observe two scenarios depending on the configuration. In the `nginx C1` configuration, `FORTDIVIDE` stops the same leak as ReMon. In the `nginx C2` configuration, `FORTDIVIDE` allows the leak because the system call that leaks the pointers is executed in SVX mode. Still, attackers can only ever leak leader pointers, meaning they can only generate addresses to gadgets in the leader variant.

Given the lack of heap diversification, we can also successfully overwrite the pointer in the hash table to launch the attack. The pointer we chose to overwrite normally points to a struct in global memory known to hold a function pointer. Our migration handler does not replicate changes to that function pointer because it normally holds a constant value, despite being stored in a writable memory region. Thus, while we can successfully corrupt the pointer in the leader, the followers do not see the updated function pointer and diverge immediately.

As an academic exercise, we updated our migration handler to replicate function pointer updates during state migration. We treat these function pointers as targeted pointer values (cfr. Section 5.1), so the monitor will translate them to equivalent follower pointers during state migration. After updating the handler, the followers successfully invoke the first gadget, but then crash when they return to the second gadget, since the rest of the ROP chain resides in a buffer that does get translated either.

We then also ran the C2 configuration with our fully automated heap scanning feature enabled (cfr. Section 5.1). This feature scans the entire heap during state migration and translates all leader pointers it finds in the Monomorphic Partition to equivalent follower pointers. As expected, the C2 attack succeeds with heap scanning enabled since the scanner updates pointers in the ROP payload. This experiment confirms that automated heap scanning is a convenient debugging aid that should not be used in production, since it compromises the security of the PMVX system.

8. Related Work

In concurrent work, Yeoh et al. also explore the idea of limiting MVX to just part of a program [52], but feature several key differences to our approach. They implement a pure in-process monitor that intercepts libc calls as RVPs. As noted in Section 2, this does not suffice to comprehensively intercept syscalls [75], and is often eschewed by security-oriented MVX systems [24]. Most notably, however, they use a clone/kill approach instead of suspend/resume, and spawn additional variants as new threads when entering MVX mode, which are initially identical to the leader apart from their stack locations. Since they utilize non-overlapping address spaces for variant diversity [17], they have to relocate all code and data mappings in each newly spawned follower, and update all pre-existing pointers to the new mappings. They implement the latter through a combination of static pointer analysis and run-time pointer scanning. Similar to us, they find that this pointer scanning is a particularly slow part of the mode transition. Due to the clone/kill approach, nearly the entire program memory must be scanned for leader pointers that require diversification, the overhead of which linearly scales with the address space usage of the program. In contrast, our resume/suspend approach requires expensive pointer scanning solely in memory regions that were updated during SVX mode, which tends to be a small subset of the total address space, and does not directly scale with the total address space usage of the program. Additionally, considering the attacks in Section 7.2, their clone/kill approach would suffer from the same downside as our automated heap scanning approach, only to a much larger extent, as it would be much harder to use finer grained diversification.

RDDR applies N-versioning in a microservice architecture [76], where MVX' inherent memory and computation overhead from its need to run multiple variants would be amplified by the number of microservices. Though its focus lies more on preventing data leaks, RDDR applies a similar idea in that it only executes critical microservices in MVX, but also does so at a different abstraction level. It uses proxies to send requests to N-versioned variants of the critical microservices, check the responses for equivalence, and merge outgoing requests to hide the existence of multiple variants, whereas the non-critical microservices run without monitoring. RDDR checks only the outbound requests for divergences, meaning that the effects of an attack have to be observable in the outbound requests. This approach does have no issues with migrating data, as each microservice is its own contained unit with its own data that only ever runs as MVX or unmonitored. We could adopt a similar idea in a compartmentalization decision where we keep certain processes in a multi-process application MVX and others SVX, but never switch them from one to the other.

9. Conclusion

In this paper, we presented Partial Multi-Variant eXecution (PMVX) as a promising alternative to complete Multi-Variant eXecution (MVX) that leverages the asymmetric security and operational requirements of different software components to apply the comprehensive and resource-intensive MVX only to those program parts that benefit from it the most. This can curb the excessive CPU time and memory consumption of complete MVX, and limit its compatibility issues or software requirements to just part of the code, removing two major obstacles towards the wider adoption of MVX technology.

We methodically analyzed the design and implementation challenges of PMVX and presented a novel design that provides application developers with efficient primitives to switch between single- and multi-variant execution while maximally preserving diversification in the variants. Our prototype implementation, `FORTDIVIDE`, includes many novel PMVX-specific optimizations that drastically reduce the overhead of such security boundary crossings.

We conclude that PMVX is a valuable option for application hardening. Still, users must carefully consider the frequency of switches between SVX and MVX, as these switches can substantially impact the protected application's performance.

Acknowledgments

We thank the reviewers and our shepherd for their feedback on improving the paper. This research is partially funded by the Research Fund KU Leuven, by the Cybersecurity Research Program Flanders, and by the Fund for Scientific Research - Flanders (FWO) under grant nr. G033520N.

References

[1] M. Miller, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape," in *BlueHat IL*, 2019.

[2] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[3] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos, "Memory errors: The past, the present, and the future," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2012.

[4] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2010.

[5] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proceedings of the USENIX Security Symposium*, 2005.

[6] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits." in *Proceedings of the USENIX Security Symposium*, 2015.

[7] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2016.

[8] H. Shacham, "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)*, 2007.

[9] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi, "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*, 2013.

[10] P. Larsen and A.-R. Sadeghi, Eds., *The Continuing Arms Race: Code-Reuse Attacks and Defenses*. Association for Computing Machinery and Morgan & Claypool, 2018, vol. 18.

[11] F. B. Cohen, "Operating system protection through program evolution." *Comput. Secur.*, vol. 12, no. 6, pp. 565–584, 1993.

[12] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*. IEEE, 1997, pp. 67–72.

[13] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2014.

[14] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi, "Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and arm," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 299–310. [Online]. Available: https://doi.org/10.1145/2484313.2484351

[15] N. Hussein, "Randomizing structure layout," 2017. [Online]. Available: https://lwn.net/Articles/722293/

[16] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz *et al.*, "Address oblivious code reuse: On the effectiveness of leakage resilient diversity." in *NDSS*, 2017.

[17] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: A secretless framework for security through diversity." in *USENIX Security Symposium*, 2006.

[18] E. D. Berger and B. G. Zorn, "Diehard: probabilistic memory safety for unsafe languages," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[19] D. Bruschi, L. Cavallaro, and A. Lanzi, "Diversified process replicæ for defeating memory error exploits," in *IEEE Performance, Computing, and Communications Conference (IPCCC)*, 2007.

[20] B. Salamat, T. Jackson, A. Gal, and M. Franz, "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)*, 2009.

[21] S. Volckaert, B. De Sutter, T. De Baets, and K. De Bosschere, "GHUMVEE: efficient, effective, and flexible replication," in *International Symposium on Foundations and Practice of Security (FPS)*, 2012.

[22] K. Koning, H. Bos, and C. Giuffrida, "Secure and efficient multi-variant execution using hardware-assisted process virtualization," in *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2016.

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

[24] S. Volckaert, B. Coppens, A. Voulimeneas, A. Homescu, P. Larsen, B. De Sutter, and M. Franz, "Secure and efficient application monitoring and replication." in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2016.

[25] M. Xu, K. Lu, T. Kim, and W. Lee, "Bunshin: compositing security mechanisms through diversification," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

[26] K. Lu, M. Xu, C. Song, T. Kim, and W. Lee, "Stopping memory disclosures via diversification and replicated execution," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2018.

[27] P. Hosek and C. Cadar, "Safe software updates via multi-version execution," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013.

[28] ——, "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)*, 2015.

[29] M. Maurer and D. Brumley, "TACHYON: Tandem

execution for efficient live patch testing," in *Proceedings of the USENIX Security Symposium*, 2012.

[30] D. Kim, Y. Kwon, W. N. Sumner, X. Zhang, and D. Xu, "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)*, 2015.

[31] Y. Kwon, D. Kim, W. N. Sumner, K. Kim, B. Saltaformaggio, X. Zhang, and D. Xu, "LDX: Causality inference by lightweight dual execution," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[32] S. Österlund, K. Koning, P. Olivier, A. Barbalace, H. Bos, and C. Giuffrida, "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)*, 2019.

[33] A. Voulimeneas, D. Song, F. Parzefall, Y. Na, P. Larsen, M. Franz, and S. Volckaert, "Distributed heterogeneous n-variant execution," in *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2020.

[34] X. Wang, S. Yeoh, R. Lyerly, P. Olivier, S.-H. Kim, and B. Ravindran, "A framework for software diversification with ISA heterogeneity," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[35] A. Rösti, S. Volckaert, M. Franz, and A. Voulimeneas, "I'll be there for you! perpetual availability in the a$\hat{8}$ mvx system," in *Proceedings of the 40th Annual Computer Security Applications Conference*, ser. ACSAC '24. Association for Computing Machinery, 2024.

[36] S. Volckaert, B. Coppens, B. De Sutter, K. De Bosschere, P. Larsen, and M. Franz, "Taming parallelism in a multi-variant execution environment," in *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2017.

[37] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal system call specialization for attack surface reduction," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1749–1766. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/ghavamnia

[38] N. Roessler, L. Atayde, I. Palmer, D. McKee, J. Pandey, V. P. Kemerlis, M. Payer, A. Bates, J. M. Smith, A. DeHon *et al.*, "μscope: A methodology for analyzing least-privilege compartmentalization in large software artifacts," in *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, 2021, pp. 296–311.

[39] M. Neugschwandtner, A. Sorniotti, and A. Kurmus, "Memory categorization: Separating attacker-controlled data," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*. Springer, 2019, pp. 263–287.

[40] S. Ahmed, H. Liljestrand, H. Jamjoom, M. Hicks, N. Asokan, and D. D. Yao, "Not all data are created equal: Data and pointer prioritization for scalable protection against Data-Oriented attacks," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 1433–1450. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/ahmed-salman

[41] S. Y. Lim, X. Han, and T. Pasquier, "Unleashing unprivileged ebpf potential with dynamic sandboxing," in *Proceedings of the 1st Workshop on eBPF and Kernel Extensions*, 2023, pp. 42–48.

[42] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1221–1238. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[43] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor: Intra-Process isolation for High-Throughput data plane libraries," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 489–504. [Online]. Available: http://www.usenix.org/conference/atc19/presentation/hedayati-hodor

[44] A. Voulimeneas, J. Vinck, R. Mechelinck, and S. Volckaert, "You shall not (by)pass! practical, secure, and fast pku-based sandboxing," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 266–282. [Online]. Available: https://doi.org/10.1145/3492321.3519560

[45] P. Kirth, M. Dickerson, S. Crane, P. Larsen, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, and M. Franz, "Pkru-safe: automatically locking down the heap between safe and unsafe languages," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 132–148. [Online]. Available: https://doi.org/10.1145/3492321.3519582

[46] K. Mitropoulou, V. Porpodas, and T. M. Jones, "Comet: Communication-optimised multi-threaded error-detection technique," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2016, pp. 1–10.

[47] L. Pina, A. Andronidis, M. Hicks, and C. Cadar, "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)*, 2019.

[48] A. Voulimeneas, D. Song, P. Larsen, M. Franz, and S. Volckaert, "dmvx: Secure and efficient multi-variant execution in a distributed setting," in *European Workshop on System Security (EuroSec)*, 2021.

[49] K. Lu, M. Xu, C. Song, T. Kim, and W. Lee,

"Stopping memory disclosures via diversification and replicated execution," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 1, pp. 160–173, 2021.

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

[51] A. Jacobs, M. Gülmez, A. Andries, S. Volckaert, and A. Voulimeneas, "System call interposition without compromise," in *54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2024, pp. 183–194.

[52] S. Yeoh, X. Wang, J.-W. Jang, and B. Ravindran, "smvx: Multi-variant execution on selected code paths," 2024, To be presented at MIDDLEWARE'24.

[53] X. Wang, S. Yeoh, P. Olivier, and B. Ravindran, "Secure and efficient in-process monitor (and library) protection with Intel MPK," in *European Workshop on System Security (EuroSec)*, 2020.

[54] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged CPU features," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, Oct. 2012, pp. 335–348. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay

[55] B. Salamat, T. Jackson, G. Wagner, C. Wimmer, and M. Franz, "Runtime defense against code injection attacks using replicated execution," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 4, pp. 588–601, 2011.

[56] J. Vinck, B. Abrath, B. Coppens, A. Voulimeneas, B. De Sutter, and S. Volckaert, "Sharing is caring: secure and efficient shared memory support for mvees," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 99–116. [Online]. Available: https://doi.org/10.1145/3492321.3519558

[57] A. Schelfhout, A. Jacobs, J. Vinck, and S. Volckaert, "Diagnosing and neutralizing address-sensitive behavior in multi-variant execution systems," in *Proceedings of the 18th European Workshop on Systems Security*, 2025.

[58] 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, "Spectre attacks: Exploiting speculative execution," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[59] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *Proceedings of the USENIX Security Symposium*, 2018.

[60] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges," in *Black Hat USA*, 2015.

[61] V. Van Der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[62] Z. Huang, T. Jaeger, and G. Tan, "Fine-grained program partitioning for security," in *Proceedings of the 14th European Workshop on Systems Security*, ser. EuroSec '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 21–26. [Online]. Available: https://doi.org/10.1145/3447852.3458717

[63] H. Lefeuvre, N. Dautenhahn, D. Chisnall, and P. Olivier, "Sok: Software compartmentalization," *arXiv preprint arXiv:2410.08434*, 2024, To be presented at S&P'25.

[64] P. Rajasekaran, S. Crane, D. Gens, Y. Na, S. Volckaert, and M. Franz, "Codarr: Continuous data space randomization against data-only attacks," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 494505. [Online]. Available: https://doi.org/10.1145/3320269.3384757

[65] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 457–468, 2014.

[66] G. Ramalingam, "The undecidability of aliasing," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1467–1471, Sep. 1994. [Online]. Available: https://doi.org/10.1145/186025.186041

[67] J. Rafkind, A. Wick, J. Regehr, and M. Flatt, "Precise garbage collection for c," in *Proceedings of the 2009 international symposium on Memory management*, 2009, pp. 39–48.

[68] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Librando: transparent code randomization for just-in-time compilers," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*, 2013, pp. 993–1004.

[69] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Proceedings of the Second European Workshop on System Security*, 2009, pp. 1–8.

[70] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM symposium on information, computer and communications security*, 2011.

[71] E. Göktas, B. Kollenda, P. Koppe, E. Bosman, G. Portokalidis, T. Holz, H. Bos, and C. Giuffrida, "Position-independent code reuse: On the effectiveness of ASLR in the absence of information disclosure," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018.

[72] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro, "Data randomization," Technical Report TR-2008-120, Microsoft Research, 2008., Tech. Rep., 2008.

[73] S. Bhatkar and R. Sekar, "Data space randomization," in *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assess-*

ment (DIMVA), 2008.
[74] S. Mergendahl, N. Burow, and H. Okhravi, "Cross-language attacks." in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2022.
[75] K. Yasukata, H. Tazaki, P.-L. Aublin, and K. Ishiguro, "zpoline: a system call hook mechanism based on binary rewriting," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 293–300. [Online]. Available: https://www.usenix.org/conference/atc23/presentation/yasukata
[76] A. M. Espinoza, R. Wood, S. Forrest, and M. Tiwari, "Back to the future: N-versioning of microservices," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2022, pp. 415–427.

Appendix
Data Availability

We provide access to all `FORTDIVIDE` source code and experiments at https://github.com/ReMon-MVEE/ReMon/releases/tag/eurosp-2025. `FORTDIVIDE` is applied as an extension to a separate branch of ReMon's public source code[3]. We have applied changes to both ReMon's cross-process and in-process monitor, but most of our changes are bundled in the `PMVEE/` subdirectory. The folder `eurosp2025/` provides a README that explains the different steps to set up and use `FORTDIVIDE` as well as a `bootstrap.sh` script that sets up `FORTDIVIDE`'s basics and the benchmarks we ran for our evaluation. This script will call ReMon's original `bootstrap.sh` and pull the correct Linux source code, apply our patch to it, compile it, and install it as a Debian package. This requires an x86 system and we recommend Ubuntu 20.04. It will then pull `nginx` version 1.23.3 and `lighttpd` version 1.4.60, and apply our small patches to it that we used insert our hooks and manual state migration.

Both servers have a script associated with them in `eurosp2025/scripts/` that will be symlinked into the `eurosp2025/nginx/` and `eurosp2025/lighttpd/` folders. These scripts provide options to compile the different versions of the benchmarks that were used, which will be described in `eurosp2025/README`. They fully take care of enabling different additional handlers, replacing hooks, and calling the different scripts that compile the data `FORTDIVIDE` requires for its execution. The microbenchmarks provided in `eurosp2025/microbenchmarks/` similarly call the relevant scripts to set up the data in their Makefiles.

To aid in running the experiments `eurosp2025/run-benchmars.sh` will provide several options to run the benchmarks. It will take care of compiling the necessary configurations of each benchmark, `FORTDIVIDE`, setting up the benchmark, and copying the results to `eurosp2025/results/`. Running `eurosp2025/processor.py` will then compile the results into the different graphs and a results report.

3. https://github.com/ReMon-MVEE/ReMon/