Secure and Efficient Application Monitoring and Replication without Kernel Patches

Bert Abrath, Lennert Franssens[1], Bjorn De Sutter, Bart Coppens

Computer Systems Lab, Ghent University, Technologiepark-Zwijnaarde 126, 9052, Gent, Belgium

Abstract

While multi-variant execution (MVX) has been demonstrated to provide precise and secretless mitigation against many classes of memory exploits at a low performance cost, achieving that low cost has so far always come at the price of a larger trusted computing base. For example, the ReMon MVX engine combines an in-process monitor with a cross-process monitor, and relies on a kernel-space broker to isolate the in-process monitor. This requires applying a special-purpose patch to the Linux kernel, which can be a significant hurdle for its use in practice.

In this paper, we present two alternative designs for that in-process monitor and its isolation. These designs build on security capabilities of modern processors and the mainline Linux kernel, without requiring any adaptation. A security analysis reveals that the novel designs are as secure as the existing ReMon design, and a performance evaluation reveals that no performance price needs to be paid.

Keywords: software security, code reuse, multi-variant execution, memory safety

1. Introduction

C and C++ are still by and large the languages of choice for systems programming because of to their unique feature sets and performance characteristics. World-wide, programmers developed huge code bases. Due to the languages' loose specifications and lack of safety checks, even the most carefully written programs often contain vulnerabilities related to undefined behavior and memory errors [28]. Hackers routinely exploit these errors to achieve privilege escalation or to get unauthorized access to confidential information [36, 38, 34, 35, 18, 19, 5, 6].

Email addresses: bert.abrath@ugent.be (Bert Abrath), lennert.franssens@gmail.com (Lennert Franssens), bjorn.desutter@ugent.be (Bjorn De Sutter), bart.coppens@ugent.be (Bart Coppens)

[1]Work on this research done as a masters student at Ghent University.

Software diversity provides one solution against memory exploits [7, 15, 25]. By randomizing code elements such as the register allocation, instruction selection, data representation, or memory layout, tools can generate many different *variants* of a program that differ syntactically but are semantically equivalent. As a result, memory exploits that can successfully compromise one variant often fail at compromising other variants. However, diversity does not provide complete protection or hard security guarantees, as its probabilistic protection can be undermined with information leakage attacks [31].

Multi-Variant eXecution (MVX) alleviates these shortcomings by running multiple program variants in lock-step at the granularity of system calls and other IO operations (e.g., through shared memory, or signals) while feeding them with identical inputs [10, 2, 3, 32, 43, 23, 40, 42, 51, 26, 16, 17, 27, 21, 24, 56, 45, 47, 39, 1]. A core principle is the use of *structurally asymmetrical* variants that have a high probability (up to 100% for certain types of exploits) of reacting differently to the same exploit payloads. The MVX monitor then detects divergences in the run-time behavior, assumes that they are the result of an ongoing attack, and takes the appropriate actions. On systems with sufficient resources, i.e., sufficient cores and memory bandwidth, MVX incurs very limited slowdowns [1].

To obtain those limited slowdowns, a performance-oriented design of the monitor is necessary. Specifically, cross-process monitoring should be avoided because performing multiple context switches (from monitored application variants to their monitor and back) for each system call introduces too much overhead. The hybrid MVX monitoring approach ReMon achieved this by monitoring system calls with an in-process monitor (IP-MON) or with a cross-process monitor (CP-MON) [42]. In-process monitoring is faster because it requires no context switches, but is potentially less secure because the monitor is not strongly isolated from the monitored program by the OS and the memory protection hardware. This might potentially allow an attacker to inject a payload into the monitored program to tamper with the monitor. The core idea is that safe system calls will be handled by the IP-MON, while potentially unsafe system calls, i.e., the ones attackers would have to rely on to mount a successful attack, are still handled by the CP-MON. Several recent advances in MVX functionality [1, 39] have built on ReMon to obtain support for more applications and against more attacks.

To enforce the security policy that specifies which system calls should be handled by which monitor, and to secure the IP-MON to a certain degree from in-process attacks, ReMon features a so-called system call broker that runs in the (Linux) kernel [42]. For each executed system call, it is this broker that allows and sets up the IP-MON to handle it, or that sends the system call to the CP-MON. However, this broker requires a patch for the Linux kernel, which is far from ideal. To ease adoption of an MVX engine like ReMon, it would be much better if it can run on top of stock kernels.

In our research, we aim specifically for that: replacing the in-kernel broker of ReMon by a solution that does not require a kernel patch. To enable this, we designed, implemented, and evaluated alternative IP-MON designs that build on several modern, now standard Linux features, combined with modern hardware support for secure execution. These include Seccomp-bpf [12], Syscall User Dispatch [13], and Intel Memory Protection Keys [11]. This paper presents the results of this research.

The main contributions of this paper are the following:

- We present two alternative designs for IP-MON that do not require kernel patches.
- We analyse their security analysis, showing that the level of security is not lowered.

- We evaluate their performance, showing that no performance price is paid with one of the designs, and highlighting the difficulty to predict performance because of interactions between the application and the operating system.
- We open source all artifacts.

The remainder of this paper is structured as follows: Section 2 introduces the necessary background information and at once discusses related work. Section 3 presents the alternative designs for the IP-MON component. Section 4 presents a security analysis focusing on the differences with IP-MON's original design in ReMon and building on ReMon's authors' analysis thereof. Section 5 presents a performance evaluation, after which Section 6 draws conclusions.

2. Background and Related Work

This section provides the necessary background on MVX, including related work and the specific MVX design for which this paper presents an alternative, as well as some background on the technologies on which our alternative design builds.

2.1. Multi-Variant eXecution

2.1.1. Asymetrical Software Diversity

The security guarantees of an MVX engine depend entirely on how they generate variants. Typically, MVX relies on *structurally asymmetrical* variants that have a high probability of responding differently to the same exploit payload. Cox et al., e.g., proposed to use compile-time transformations to generate variant binaries with non-overlapping address spaces [10]. This setup thwarts any code-reuse or data-oriented attacks that rely on payloads containing absolute memory addresses. Several other MVX engines achieve the same results by relying on run-time transformations to generate asymmetrical variants of a single binary [42, 26]. More examples of effective variant generation techniques exist [32, 42, 26, 23, 56, 45, 47, 44].

2.1.2. Rendezvous Points

A key task of the MVX monitor is to observe variants and detect when their execution behavior divergences. To that extent, security-oriented MVX engines suspend variants when they reach so-called *rendezvous points* (RVPs). Once all variants arrive at an RVP, the monitor checks their states for equivalence and takes a corrective action when the states do not match. If the states match, the monitor resumes the variants. Most MVX systems use system call entry and return points as the RVPs, under the assumption that system calls are the mechanism on which exploits rely to harm a system from within an exploited application [10, 32]. Vinck et al. extended the RVPs to shared memory (SHM) accesses [39], because those could also allow an attacked application to execute inter-process communication (IPC), i.e., to interact with the external system to harm it.

RVPs also play a crucial role in I/O replication. When the leader variant reaches the return RVP of an input operation such as a `sys_read`, the monitor records the result of that operation and replicates it to the follower variants.

2.1.3. Relaxed Rendezvous Points

Suspending variants at every system call entry and return point can be detrimental to their run-time performance. Some MVXs hence support *relaxed RVPs* to exempt certain system calls from the strict lockstepping requirement [17, 42, 23]. When the leader variant reaches a relaxed RVP, the monitor records its state and, potentially, system call results, but allows it to continue its execution immediately. When the follower variants reach the same relaxed RVP, the monitor uses the recorded data to perform state checking and/or I/O replication. If a follower reaches a relaxed RVP before the leader does, it waits for the leader to reach the RVP and record the necessary data.

In security-oriented MVX designs, all potentially dangerous system calls need to be surrounded by regular RVPs, such that the MVX engine can guarantee that no harmful actions can be performed unless all variants get compromised simultaneously (which is mitigated by the deployed diversity as discussed above). MVX designs for other purposes, such as software testing [17] and safe updates [29], can use relaxed RVPs everywhere. This allows them to tolerate expected divergences [30]. Unexpected divergences can still be detected, but only after the divergent actions have completed.

2.1.4. Security versus Efficieny

Prior research explored many designs for MVX monitors with different security-performance trade-offs, in an attempt to meet multiple, conflicting goals:

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

The original MVX design by Cox et al. executed its monitor in kernel space [10]. In-kernel monitors provide strict RVP enforcement, strong monitor isolation, and low invocation overhead, but have a substantial TCB footprint. Several later designs run the monitor as a standalone process in user space and rely on the `ptrace` API to intercept system calls [40, 32, 4]. These designs also provide strict RVP enforcement and strong memory isolation. Unlike in-kernel monitors, however, they have a minimal TCB footprint and they incur high run-time overhead since they require context switching for every RVP [17, 42].

VARAN uses selective binary rewriting to intercept system calls in user space and relies on an in-process monitor to avoid costly context switches [17]. This design provides the lowest monitor overhead, but does not isolate the monitor at all, and it allows compromised variants to bypass RVPs [42]. Some approaches at more improved system call interception mechanisms such as zpoline also require binary rewriting [54]. MvArmor [23] and MonGuard [48] enforce isolation of the in-process monitor component using

hardware virtualization or isolation extensions [8]. MvArmor incurs low system call interception overhead but requires substantial additions to the TCB. MonGuard similarly incurs low overhead by relying on Intel's MPK extension. sMVX extends MonGuard to limit the amount of code paths in the program for which multiple variants are required, reducing the replication overhead [55].

ReMon is a hybrid design that combines a `ptrace`-based, cross-process monitor (CP-MON) with an efficient in-process monitor (IP-MON) [42]. ReMon's CP-MON provides strict RVP enforcement for regular RVPs, while its IP-MON implements equivalence checks and replication for relaxed RVPs. The IP-MON relies on information hiding to isolate both itself (i.e., its code) and the replication buffer (RB) it uses for replication (i.e., its data), from potentially compromised variants. ReMon does, however, require a patch to the kernel, which is a major barrier towards adoption of such an MVX design.

2.1.5. Implicit Progam Inputs

A big challenge in the design and implementation of an MVEE is ensuring that the variants behave equivalently if they receive inputs from other sources than the system call interface. Prior research identified several such sources, including the CPU's time stamp counter and random number generator [32], the virtual system call interface [17], asynchronous signal delivery [33], and SHM IPC [3]. Researchers also pointed out that the behavior of many programs depends on run-time execution properties such as their address-space layout [43] or the order in which threads observe modifications other threads make to shared state [41]. These execution properties could be viewed as implicit program inputs. Reading input from any of these sources can cause so-called benign divergences, where the variants appear to behave differently despite not being attacked. Thus, to support the widest range of programs, an MVX must guarantee that variants read identical input from all of these resources.

2.1.6. ReMon

As our goal is to design a system that offers the same security/efficiency balance as ReMon, but without requiring a kernel patch, we now discuss ReMon's design in more detail. It consists of three components: a security-oriented CP-MON for security-sensitive system calls; an IP-MON for non-security-sensitive system calls; and a broker component in the kernel, which decides which system calls are to be handled by which monitor [42]. The broker makes this decision according to a configurable security policy, which is also known to the IP-MON. The policy in essence is a partitioning of all system calls into those that are monitored and those that are not, i.e., those that will be handled by the CP-MON vs. those that will be handled by the IP-MON. Furthermore, the policy also allows to specify conditional monitoring for system calls. In that case, the policy specifies which condition should be met by the dynamic arguments of a specific system call invocation to determine whether that invocation should be monitored or not, i.e., whether it should be handled by CP-MON or IP-MON.

The CP-MON is also responsible for launching the multiple variants of the application. As CP-MON is a separate process, it attaches to the launched variants using the ptrace interface to ensure it can intervene in all system calls executed by the variants.

When variants are launched, CP-MON injects an IP-MON into each variant's address

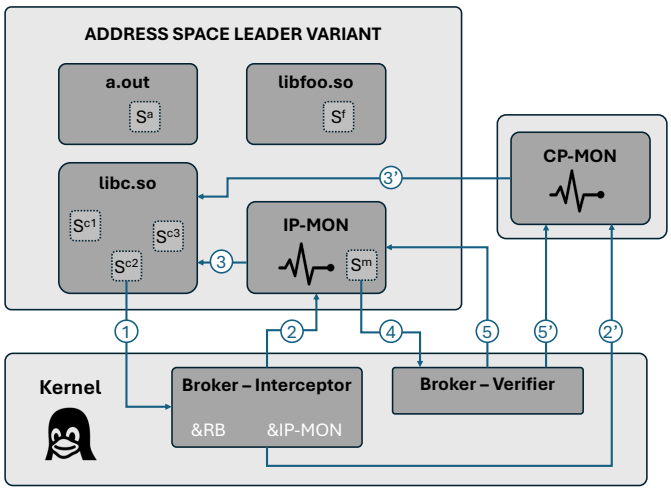

Figure 1: The original ReMon design. The components making up the system are shown in dark gray boxes. Light gray boxes show how they are partitioned between separate user-space processes (variant and CP-MON) and the kernel. System call instructions are shown in small boxes with an S inside. Text in white indicates where secrets, namely addresses of security-sensitive code and data, are stored persistently.

space.[2] In each variant, the IP-MON then registers itself with the CP-MON, and receives a replication buffer. This buffer is shared between the different variants, and used by their IP-MONs to communicate with each other.

Next, the IP-MON registers itself with the in-kernel broker, passing along the security policy, and a pointer to the replication buffer. Afterwards, it removes this pointer, so no reference to the replication buffer remains in the variant's address space. The addresses of both this buffer and IP-MON itself are from then on secrets only known to, and stored persistently by, the in-kernel broker. In the leader variant, IP-MON from here on operates in leader mode; in the follower variants it operates in follower mode. Notice that all initialization of IP-MON happens before the application itself starts executing, so it is independent of the user-provided inputs and cannot be hijacked by an attacker.

We will use Figure 1 and refer to its numbered arrows to explain how system calls are handled in ReMon. In the figure, system call instructions are shown with small boxes with an S inside. Practically, it does not matter from which part of a program a system call is made, they are all handled exactly the same. In this running example, we will assume that the system call S^{c2} is invoked from within a standard C library function, i.e., from within the libc.so binary of which the code and statically allocated data were mapped into each variants' address space. The figure shows only the leader variant, but of course one or more follower variants are also executing in parallel. As long as the variants behave equivalently, if IP-MON is invoked in one of them, then it is invoked in

[2]Throughout the paper, we describe the internal operation of different MVX components as if the variants are single processes. In multi-process applications, each additionally launched application process is handled in the same way as the main process. A new, separate CP-MON attaches to the variants of that new application process, and injects new IP-MONs into each variant. The new CP-MON and the new IP-MONs behave identically to those of and in the main application process variants.

all of them.

Whenever a system call is made ①, it is first intercepted by the in-kernel broker, which decides whether the call is to be monitored or might be unmonitored. Monitored system calls are forwarded ②' to CP-MON, by allowing `ptrace` to intercept it. CP-MON then handles all replication and checking. If the checks are successful, CP-MON then returns control back to the variant ③' to allow it to continue its execution. System calls that might be unmonitored are redirected ② to IP-MON by letting the kernel broker change the variant's instruction pointer to IP-MON's entry point (i.e., `&IP-MON`). Before returning to IP-MON's entry point, i.e., before allowing the variant to continue execution at the newly set instruction pointer, the broker also provides the address of the replication buffer (i.e., `&RB`) in a register known to IP-MON, as well as a one-time secret authorization token in another known register. IP-MON has been implemented in such a way that it never writes the key and the token to memory, such that they cannot leak to the application itself.

For conditionally unmonitored system calls, IP-MON can then decide whether the system call is to be monitored or not based on the arguments. If the system call is unmonitored, all variants' IP-MONs verify that all of them want to make the equivalent system call and, if this is indeed the case, then make the system call again ④, now passing along the authorization token. The in-kernel broker verifies that the address of the variants' `syscall` instruction (S^m) is located inside of each IP-MON and that the authorization token is correct. If that is the case, the system call finally gets executed (in the leader alone, or in the leader and the followers, depending on how the system call needs to be replicated), and its result is returned ⑤ to IP-MON to perform the necessary replication, after which control is returned ③ to the code that originally invoked the system call, in this case the instruction following (S^{c2}) in `libc`. Otherwise, the system call is still forwarded to CP-MON ⑤'.

The security analysis by Volckaert et al. [42] makes it clear that the in-kernel broker allows ReMon's design to function securely and efficiently. However, it relies on a custom patch for the Linux kernel, which hinders its adoption. Below, we discuss three technologies on which alternative monitor designs can build to deliver the same features as ReMon, but without a kernel patch.

2.2. Seccomp-bpf

Seccomp [12] is a Linux kernel feature that allows a process to restrict its own ability to make system calls. Filter mode is a specific seccomp mode that allows the process to specify a filter for system calls, expressed as a Berkeley Packet Filter or BPF. This filter is then executed for every system call the process attempts to make, and decides whether the system call should be allowed, denied, or some other action needs to be taken.

Seccomp-bpf has been used for system call interposition in the context of sandboxing without requiring root privileges [22]. In the context of MVX, it has only been used in VARAN [17], a reliability-oriented monitor rather than a security-oriented one. VARAN does not enforce lockstep execution, and its IP-MON is not shielded off from the application code with which it resides in the variants. Its primary goal is to run evolved variants of the same application side by side, e.g., one with and one without a new patch applied to them, to test that the patch does not introduce unintended side effects. Such patches may introduce benign changes in the invoked system calls, so VARAN needs to allow variants to diverge in their system calls, contrary to how security-oriented MVX

solutions such as ReMon consider each divergence as a symptom of an attack. VARAN uses seccomp-bpf to support application-specific system call rewrite rules, i.e., rules that specify which changes in system call patterns are allowed between the variants of a specific application. This use of seccomp-bpf hence serves a completely different purpose ot the one presented in Section 3.1.

Once installed, a seccomp filter cannot be removed from a process—not even when the process does a `fork` or an `execve`. This is important for multi-process applications in which a main process launches additional processes, because it implies that the same filter will be applied to all those processes.

When a seccomp filter is executed, it returns the action to be taken as a result. If a process has multiple seccomp filters, they are all executed, but only the action with the highest precedence is taken. We briefly discuss the actions that are of interest for this paper, in decreasing order of precedence:

- `SECCOMP_RET_KILL_PROCESS`: the process is immediately terminated.
- `SECCOMP_RET_ERRNO`: do not execute the system call, and return a portion of the filter's return value to user space as `errno`. While technically a seccomp filter can have a 16-bit return value, the Linux kernel limits this value to a 12-bit `errno`.
- `SECCOMP_RET_TRACE`: the kernel will attempt to notify a `ptrace`-based tracer prior to executing the system call. If a tracer exists, it now decides how to handle the system call. If there is no tracer, the system call is not executed.
- `SECCOMP_RET_ALLOW`: allow the system call to be executed.

2.3. Syscall User Dispatch

Syscall User Dispatch [13] (**SUD**) is a Linux kernel feature that allows a thread to selectively intercept and control the handling of system calls based on their originating address. Its intended use cases are compability layers such as Wine, as it allows for efficient emulation of system calls. Jacobs et al. [20], for example, combined SUD with binary rewriting to achieve efficient system call interposition. In the second alternative design for IP-MON, that will be presented in Section 3.2.2, we will also use SUD, albeit without requiring binary rewriting. Instead we will apply a minor refactoring of system call code in the C library to achieve the necessary efficiency.

When SUD is enabled, only system calls originating from a specific memory region are directly allowed. The kernel intercepts all system calls originating from outside that known region, and redirects them back to the process in the form of a `SIGSYS` signal. An appropriate signal handler can then provide the necessary emulation. SUD can be quickly disabled (and enabled again) from the thread itself by changing the value of a memory location, called a flip switch. This address is registered with the kernel, and its value controls whether SUD is enabled or not.

2.4. Memory Protection Keys

Intel Memory Protection Keys [11] (MPK) is a hardware feature that allows a process to efficiently manage access permissions for memory regions, without making any system calls. A single user-space instruction (`WRPKRU`) suffices to change a region's permissions

(e.g., make that region writable). MPK can be used to efficiently implement in-process isolation. For example, a sensitive data structure can be made accessible only when it has to be accessed, and be made inaccessible right after.

Early proposals for the use of MPK for Userspace (dubbed PKU) have been found incomplete by Voulimeneas et al. [46], and hence susceptible to attacks. Their Cerberus solution relies, amongst others, on monitoring, intercepting, and blocking system calls. In the alternative IP-MON design that we will propose in the next section, that functionality is naturally provided by the CP-MON component.

For contexts such as browsers, library OSes, and system emulators that rely on sandboxes and in-process isolation to isolate untrusted components securely, Yang et al. proposed so-called nexpolines that combine MPK with seccomp or SUD as an alternative to ptrace, because of nexpolines' higher efficiency [53, 52]. Like this paper does for MVX, they noted the advantage of not requiring kernel modifications. The alternative IP-MON designs proposed in this paper offer the same advantage over ReMon's in-kernel broker. The isolation of IP-MON in one of the two alternative designs proposed in this paper (the alternative proposed in Section 3.2.2) has been inspired by nexpolines' design.

3. IP-MON without Kernel Patch

Our goal is to replace ReMon's in-kernel broker with a solution that does not require any modifications to the kernel. In the original design IP-MON and the broker implement a configurable security policy, deciding which system calls are to be monitored and which to be unmonitored. The broker also guards two secret addresses that are only provided (temporarilly) to user space when handling unmonitored system calls: the address of the replication buffer, and IP-MON's entrypoint.

Crucially, placing ReMon's broker in the kernel separated it from the application through a privilege boundary. Attackers could thus not directly influence the security policy, and the locations of IP-MON and the replication buffer are hidden from them. As we want to provide an alternative to the in-kernel broker without compromising security, our replacement design also requires some privilege boundary between the application and the policy implementation, as well as between IP-MON and the rest of the application in user space. Access to IP-MON thus needs to go through some sort of call gate.

In our replacement design we chose to implement the security policy as a seccomp filter. Seccomp is a natural fit for the problem, and we immediately achieve the required privilege boundary between the application and the policy implementation.

In our new design, the address of the replication buffer will be stored persistently in IP-MON itself, rather then in the kernel. This makes it even more important to place a privilege boundary between IP-MON and the rest of the application. For the call gate that guards this boundary we came up with two design alternatives. We will now explain all design aspects and alternatives in detail, focusing on their functional equivalence to the original design. We will analyze their security aspects later on, in Section 4.

3.1. Seccomp-Based Security Policy

During its initialization, IP-MON translates the specified security policy into a BPF program, and installs it as a seccomp filter in the kernel. Every time the variant process makes a system call, this filter executes. It can then decide whether the system call is to

be monitored or unmonitored, based on the type of system call and the address of the `syscall` instruction.

If the seccomp filter decides that the system call is to be monitored, it immediately forwards the system call to CP-MON by means of a `SECCOMP_RET_TRACE` action. CP-MON then handles the execution and replication of the system call, in exactly the same way as is done in ReMon.

If the seccomp filter decides that the system call is unmonitored, it allows the system call to execute by means of the `SECCOMP_RET_ALLOW` action. IP-MON will from then on handle all equivalence checking in the variants, and handle the necessary replication, just like it does in ReMon.

The seccomp filter is defined such that the latter decision is only made when the policy allows for unmonitored execution, and when the system call was invoked from within IP-MON, and even from a single address in IP-MON, which we will denote S^u. Consequently, system calls originating from other code locations (in libc.so, in other libraries, or in the main application code section) are never executed as unmonitored system calls. Importantly, since each variant has a separate IP-MON and a separate seccomp filter, this address S^u can differ from one variant to another. For each variant, the address S^u is hardcoded into the variant's seccomp filter at initialization, and can hence never change afterwards. In multi-process applications, this implies that a variant's IP-MON's system call instruction needs to remain at S^u in all the variant's processes created through forking or execve.

3.2. Call Gate

As described above, system calls can be allowed to execute unmonitored, i.e., be checked and replicated in IP-MON, when they are invoked from S^u in IP-MON. This means that access to that address S^u (i.e., to IP-MON) from within the application should be protected through a mechanism that includes a privilege boundary. We refer to that mechanism as a call gate, and created two different call gate designs.

3.2.1. Seccomp-Based Call Gate

In this first design alternative, the call gate is implemented by the seccomp filter, which operates in kernel space, and a short instruction sequence starting at address S^i in user space, i.e., somewhere in the application. Besides S^u, the seccomp filter also recognizes S^i. Upon invocation of a potentially unmonitored system call from S^i, the seccomp filter will provide the secret address of IP-MON to the call gate code, which can then transfer control to it.

Figure 2 gives an overview of all components involved. The variant process is traced by the CP-MON process, and IP-MON is present in its address space. Other noteworthy components in the variant's address space are `libc.os`, other libraries such as `libfoo.os`, and the call gate. We describe the control flow for system calls originating from `libc` step by step. The reader can follow along on the figure using the numbered arrows.

Once IP-MON is initialized, all the system calls that `libc` wants to make are rerouted ① through the call gate. How this rerouting is implemented will be discussed later.

The call gate instruction sequence starts with executing ② the `syscall` instruction at S^i for the rerouted system call. If this system call is a monitored one, the seccomp filter will perform a `SECCOMP_RET_TRACE` action ③, and the system call will thus immediately

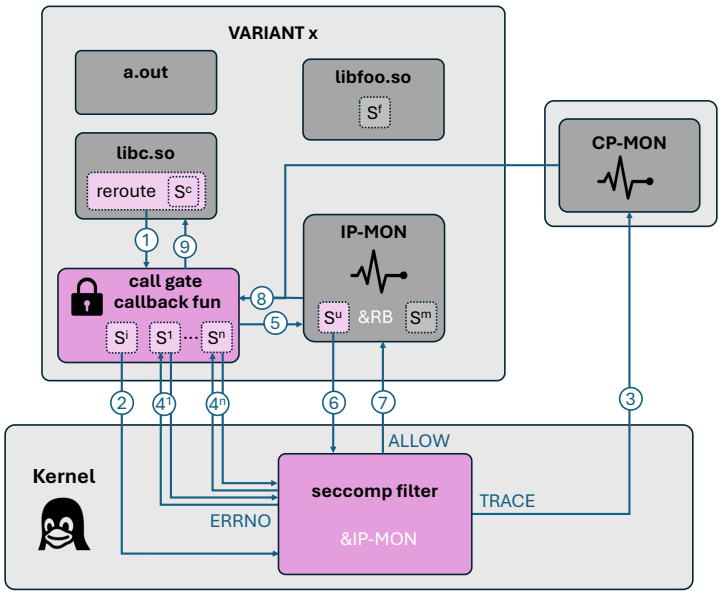

Figure 2: The seccomp-based design, using the same visualization template as Figure 1. New components compared to the existing ReMon design are colored in pink. The additional text in blue denotes the actions taken by the seccomp filter.

be forwarded to CP-MON, without any IP-MON involvement. If the filter instead decides that the system call can be unmonitored, it performs the `SECCOMP_RET_ERRNO` action, returning the `MAX_ERRNO` value to user space. This value cannot be mistaken for an actual errno, and thus user space now knows to execute the syscall sequence.

Next, the seccomp filter transfers IP-MON's secret entrypoint to the call gate. Transferring an entire address requires multiple `SECCOMP_RET_ERRNO` actions $\textcircled{4^1} \cdots \textcircled{4^n}$, because a seccomp filter can return at most 12 bits at once through the `SECCOMP_RET_ERRNO` action. For that reason, a sequence $S^1..S^n$ of `syscall` instructions in the call gate code fragment are executed, each of which allows the transfer of additional bits.

A complete address is 64 bits. In theory, therefore, six `syscall`s suffice to transfer a complete address. As the upper 16 bits of an address are currently sign-extended on the x86-64 architecture, four `syscall`s suffice in practice. By aligning the entrypoint to the page size, we can fix the 12 least significant bits of the address, and three `syscall`s suffice. This last step reduces security, however.

After the transfers of all the necessary bits are completed, the call gate fragment ends by directly invoking ⑤ IP-MON by means of the received address. IP-MON discards the address immediately, such that it cannot leak to any other code fragment in user space, and such that for any additional system calls that the application wants to execute later, it will need to pass through the call gate again.

IP-MON already knows the address for the replication buffer, so it can then perform the necessary verification checking and replication for the requested system call. When IP-MON decides to implement the requested system call via an unmonitored system call, it executes ⑥ the `syscall` instruction S^u. The seccomp filter recognizes this address as

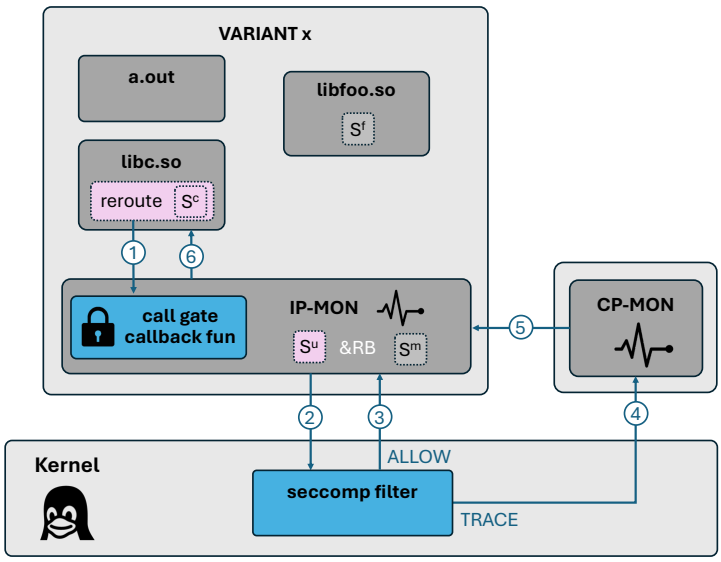

Figure 3: The user-space-based design, using the same visualization template as Figures 1 and 2. Changes in this design compared to ReMon's design and the seccomp-based design from those earlier figures are colored blue.

special, and after verifying this type of system call can indeed be unmonitored, allows the system call by means of the `SECCOMP_RET_ALLOW` action ⑦.

When alternatively IP-MON would still decide to implement the requested system call via a monitored system call, it will execute the `syscall` instruction S^m instead of S^u. Like it does for any other system call not originating from the call gate or S^u, the seccomp filter will then respond with a `SECCOMP_RET_TRACE` action, resulting in the system call being handled by CP-MON. This is the case for calls invoked at S^m, but also, e.g., S^f in `libfoo.so` or S^c, the single system call instruction that is still present in `libc` to make that library function correctly when no IP-MON has injected the call back function. That makes it possible to use the same libc.so irrespective of whether the application is executed without MVX, under an MVX engine with only CP-MON enabled (like GHUMVEE [40]), and under a ReMon-like MVX with both a CP-MON and an IP-MON [43, 42].

After CP-MON or IP-MON handled the system call and performed the necessary replication, they transfer control back to the call gate ⑧, after which it will be returned ⑨ to libc.so, to allow the variant to continue its execution.

3.2.2. User-Space Call Gate

While the above call design is secure, as will be discussed later in Section 4, the overhead of the system call sequence starting at S^i can be considerable, as will be measured in Section 5.2 for a syscall-heavy workload. That is the main reason for considering an alternative call gate design that builds on MPK and SUD.

In this second design, the address of IP-MON is no longer a secret passed from kernel space to allow the call gate to transfer control to IP-MON. Instead, the call gate, including the transfer to IP-MON, is implemented entirely in user space.

In this design, however, only invocations of IP-MON that have passed through the call gate will have access to the replication buffer and will have the ability to make unmonitored system calls. If the variant process makes a system call without going through the call gate, it will always be forwarded to CP-MON.

Figure 3 gives an overview of all components involved: The variant process is again traced by the CP-MON process, and IP-MON is present in its address space. Other components of note are `libc`, and possibly other libraries. We will explain the control flow for system calls originating from `libc` step by step. The reader can follow along on the figure using the numbered arrows.

Once IP-MON is initialized, all the system calls that `libc` wants to make are rerouted ① to IP-MON itself, more specifically to the call gate inside IP-MON, in which IP-MON's access to the replication buffer is enabled. To do so, we employ two technologies: MPK and SUD. We use MPK to place the replication buffer in a separate security domain, so that the `wpkru` instruction in the call gate has to be executed before it becomes accessible. We use SUD to create a system call switch in the replication buffer, which the call gate has to activate before system calls can be made. If any system call is invoked while the switch is off, a `SIGSYS` signal is sent to the variant process, which forces an intervention by the tracer, i.e., CP-MON. When the switch is on, we simply allow all system calls, i.e., from anywhere in user-space memory. This hence allows the system call that currently needs to be executed, which is the sole goal of enabling the switch. Allowing other system calls as well is not strictly necessary, but it also poses no risks. As the call gate is already executing for the current system call and as the switch will be disabled immediately after that calls execution, no other system call can even try to execute. The only reason for allowing all system calls then is that this is the SUD option with the least overhead.

Note that in our design, MPK protects only the replication buffer data, but not the buffer's header (or any other data structures). The reason is that the buffer's header contains the state of the SUD system call switch, which the kernel needs to check, and which can therefore not be protected with MPK.

After passing through the call gate, IP-MON has access to the replication buffer and can make system calls. If IP-MON decides the system call is unmonitored, it executes ② the `syscall` instruction S^u. Otherwise, IP-MON can also make monitored system calls, from any other address such as S^m. Regardless of what decision IP-MON makes, the seccomp filter still enforces the security policy. Only system calls that originate from S^u and whose type might be unmonitored are allowed through the `SECCOMP_RET_ALLOW` action ③, the rest are simply forwarded to CP-MON through the `SECCOMP_RET_TRACE` action ④.

After performing the necessary actions, and before returning control to the application, IP-MON switches off the ability to make system calls, and makes the replication buffer inaccessible again through the `wpkru` instruction.

Any `syscall` instruction executed outside of IP-MON (e.g., S^f in `libfoo` or S^c in `libc`) results in a `SIGSYS` signal, and will thus be handled by CP-MON.

Anytime CP-MON handled a system call, it returns control after having performed the necessary replication and bookkeeping. In case the system call originated from S^m, return is controlled to IP-MON ⑤, which will simply return control to `libc` ⑥ to continue the execution of the program. In case IP-MON handled the system call unmonitored, i.e., when it was executed via S^u and control was returned to IP-MON ③, IP-MON performs the replication and bookkeeping internally, before returning control to `libc`.

The user-space call gate introduces less overhead than the seccomp-based gate, as will be confirmed experimentally in Section 5.2. Its design is also less complex to implement. Moreover, it does not abuse the seccomp mechanism for something it was not designed for, namely returning a secret address to the application. The only potential downside is its relience on MPK, an architecture-specific hardware extension. This makes porting it to different architectures non-trivial. Other architectures have features and extensions that can be leveraged to achieve similar effects as MPK: ARMv7 has Memory Domains [37], and ARMv8 has the Memory Tagging Extension and Pointer Authentication extensions that can be combined [14], as well as the ARMv8 Privileged Access Never feature and Load/Store Unprivileged instructions [50].

3.2.3. System Call Rerouting

We adapted both the CP-MON and IP-MON components to function without a kernel patch. To easily reroute system calls to the call gates, we made some extra changes to the MVX-enabled version of glibc, the version of the standard C library (i.e., `libc`) used in our implementation.

Concretely, in the glibc source code, we have patched all `syscall` instructions by means of macros, thus replacing them by invocations of a single, central function that can make all system calls through a single `syscall` instruction. As a result, this central system call function holds the only `syscall` instruction remaining in the libc binary. Before executing that instruction, the function checks whether a callback function has been installed by IP-MON (by simply checking a pointer). If this is not the case (i.e., when no IP-MON has been loaded), the `syscall` instruction is executed, and the system call will be monitored by CP-MON. In the opposite case, when a callback function has been installed by IP-MON when it got loaded, this callback function is invoked instead of executing the `syscall` instruction. It is this callback function that then invokes the call gate to execute the system call. In both cases, after the system call has been executed, including all the necessary equivalence checking and replication by either IP-MON or CP-MON, whichever of the two monitors handled the system call can then let the application continue execution in the central system call function.

Importantly, when the patched C library is used by an application without MVX monitoring, it still functions correctly and equivalently to the unpatched library. So applications can use it when they run without MVX protection, as well as with MVX protection, with CP-MON only or with both a CP-MON and an IP-MON.

4. Security Analysis

In this section, we analyze whether the security of our seccomp-based design is equivalent to ReMon's original design. We build on the security analysis of ReMon [42], and only investigate in detail the aspects that differ.

Just like in the original ReMon design, monitored system calls are still handled by CP-MON in our alternative designs, and thus as hard to abuse for any attackers. Somehow confusing or subverting the seccomp filter so that it allows a monitored system call instead of forwarding it to CP-MON is impossible. Seccomp filters cannot be removed, nor can their actions be overridden by newer filters with less secure actions: the seccomp filtering itself is handled by the kernel, which is part of our TCB even without any IP-MON.

In the remainder, we hence analyze the case of attackers making malicious system calls that could be unmonitored. It is important to note, though, that "as long as the attacker executes unmonitored calls only according to a given relaxation policy, those capabilities by definition pose no significant security threat: unmonitored calls are exactly those calls that are defined by the chosen policy to pose either no security threat at all, or that pose an acceptable security risk" [42].

We consider two attack scenarios for system calls that could be unmonitored: manipulation of the replication buffer, and bypassing IP-MON's verification checks on conditionally unmonitored system calls.

Replication buffer manipulation When executing unmonitored system calls, the leader variant can run ahead of the followers. It can thus make system calls before the follower's IP-MON has verified it wants to make the same system call. An attacker could manipulate the replication buffer to delay or tamper with the system call verification, allowing them to execute more system calls. In ReMon, an attacker could not easily access the replication buffer as its address was only known to the in-kernel broker, and the attacker could only find it through random guessing or side-channel attacks. In our designs based on seccomp filters, IP-MON knows the replication buffer's address. As attackers do not know the variants' addresses of IP-MON, however, they are once again limited to random guessing or side-channel attacks. As a result, the more entropy available in the variants' IP-MON addresses, the better.

Bypassing IP-MON's verification For conditionally unmonitored system calls, IP-MON decides whether the system call is to be monitored or not based on its arguments. Because comparing arguments between variants is impossible to implement in a seccomp filter—just like it was impossible to implement in the in-kernel broker—the verification of this policy is only implemented in IP-MON. If attackers could subvert this logic, they could execute an unmonitored system call whereas according to the policy it should be monitored. If IP-MON is properly invoked through its entrypoint, verification checks will happen correctly. However, if an attacker has compromised a variant, **and** found IP-MON's address, they could attempt to jump into IP-MON right after these checks. In the original design this attack scenario was neutered through the use of an authorization token provided by the in-kernel broker. Although our design has no such token, our user-space-based call gate guards IP-MON's ability to make system calls. In order to make a system call, control flow has to pass through the call gate, once again rendering this attack impossible.

We conclude that the security of our seccomp-based designs is equivalent to the original design if the user-space call gate is used.

5. Performance Evaluation

We evaluated the performance overhead of the different call gate designs on the SPEC benchmarks, as well as on the nginx web server. All of these benchmarks were compiled with LLVM 18.1.8 at optimization level -O2. We established a baseline performance for the individual benchmarks by running them as stand-alone programs, i.e., without any MVX. We then measured the performance of these benchmarks for four MVX configurations, all of which are based on ReMon's publicly available MVX engine [42]:

1. no IP-MON at all, only CP-MON;

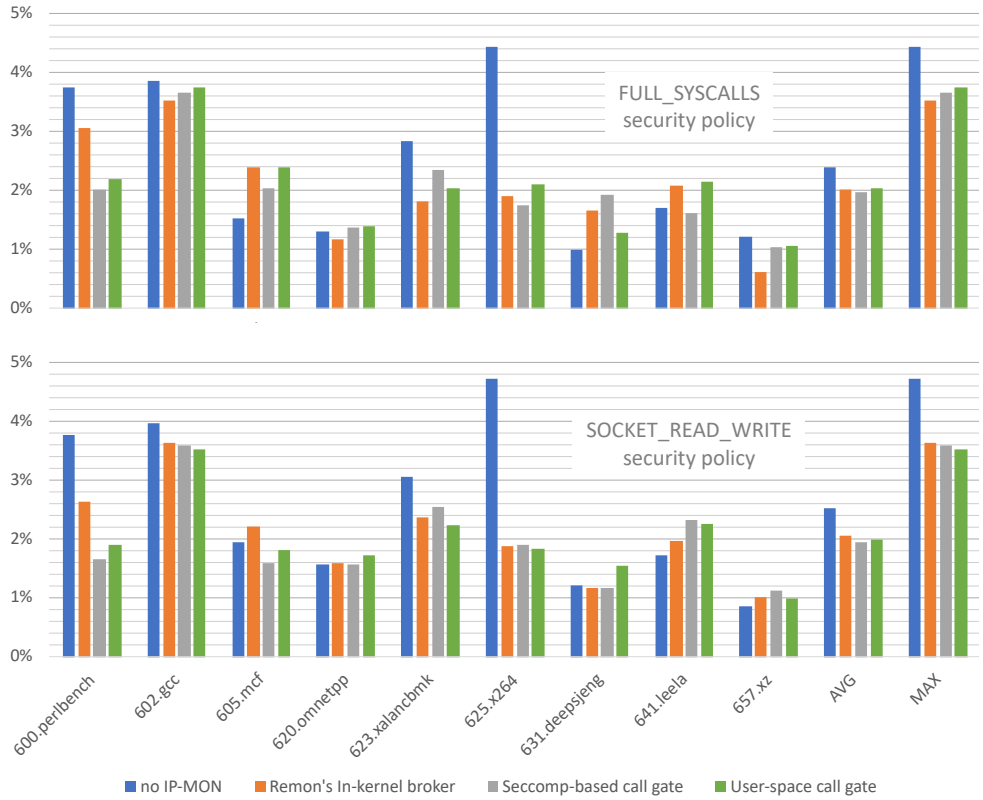

Figure 4: Performance overhead relative to unprotected program execution for the four evaluated MVX configurations under the two evaluated security policies.

2. CP-MON + IP-MON using ReMon's in-kernel broker;
3. CP-MON + IP-MON using our seccomp-based call gate;
4. CP-MON + IP-MON using our user-space call gate.

The diversification employed by the MVX engine is disjoint code layouts [40], i.e., the same diversification employed in the original ReMon evaluation [42] and in other publications using the same engine [41, 39, 1].

We performed our measurements on an AMD Ryzen Threadripper PRO 7985WX with 64 cores, 256MB cache, and hyper-threading enabled, with 512GB of RAM, running Ubuntu 24.04 and Linux kernel 6.8. For the second configuration we applied ReMon's patch to the Linux kernel that implements the in-kernel broker.

5.1. SPEC 2017

We performed our evaluation using all C and C++ programs[3] in the integer SPEC 2017 benchmark suite [9]. No code changes were made to these benchmarks to run them

[3]We omit the Fortran benchmark 548.exchange2 because LLVM lacks a Fortran front-end.

under the monitoring of the MVX engine. We ran each benchmark 15 times on the reference inputs, exclude outliers (those runs more than 2.5σ away from the average) and computed the average of the remaining run times.

We ran measurements using two security policies. First, of the security policies predefined by ReMon's authors [42], the SOCKET_RW_LEVEL policy[4] is the least restrictive one, i.e., in which the largest set of system calls are unmonitored and hence handled by IP-MON. It is hence their policy for which the change of IP-MON's design we propose can have the largest performance impact.

The second security policy we used is the FULL_SYSCALLS policy, in which all system calls are handled by IP-MON (except for two memory allocation system calls that are not supported by ReMon's IP-MON). We included this policy because it stresses IP-MON even more than the SOCKET_RW_LEVEL policy.

Figure 4 shows the measured performance overhead for the two policies for all four evaluated MVX configurations compared to the baseline.

While the configuration without IP-MON has an average overhead of around 2.5%, the three configurations with IP-MON have an average overhead of around 2%. On average, as well as worst-case, our alternative designs are about as efficient as ReMon's design with its in-kernel broker. That is the major conclusion of our measurements.

Which of the three designs is most efficient varies for individual benchmarks and for the implemented security policies. Only for one benchmark, 600.perlbench our alternative IP-MON designs are consistently faster by about 1%. For all other benchmarks, the differences in performance between the three IP-MON designs are much smaller, and not consistent accross security policies. As these differences can be due to uncontrolled parameters (e.g., for one benchmark, one design accidentally happens to achieve better cache behavior for one policy, and another design accidentilly happens to do so for the other policy) no meaningfull conclusions can be drawn about those variations.

As the SPEC benchmarks were designed to evaluate compute-intensive performance, the picture might be different for benchmarks that are more I/O-intensive.

5.2. Nginx

As an I/O-intensive benchmark, we chose the Nginx web server (version 1.18.0). To make this multi-process application MVX-compatible, i.e., to avoid benign divergences related to synchronization, we ran an atomicizing compiler pass [41] and we annotated the types of several variables in the source code to provide information to the pass. Concretely, we changed 4 lines, marking 4 variables as not being used in synchronization operations. These changes are identical to those used in existing evaluations MVX designs with Nginx [41, 39, 1].

We used wrk [49] to benchmark the performance of the server in two network setups: on a separate machine connected to our server through a gigabit Ethernet link, and on localhost. The localhost setup is rather unrealistic, but serves to stress-test the server software. The system call-rate increases unrealistically in this setup, comparable to how system calls would be executed in a system-call heavy microbenchmark. In both setups,

[4]The system calls handled by IP-MON in the SOCKET_RW_LEVEL policy are the following ones: Read only calls that do not operate on file descriptors and do not affect the file system, read and write calls on regular files, pipes, file descriptors, and sockets; read-only calls from file system; write calls on process-local variables.

Connection	Unprotected	protected with MVX with FULL_SYSCALLS security policy							
		No IP-MON		IP-MON					
				in-kernel broker		seccomp call gate		user-space call gate	
Throughput (requests per second)									
localhost	98236	9716	*-90.1%*	79174	*-19.4%*	38137	*-61.2%*	79073	*-19.5%*
gigabit	26858	9720	*-63.8%*	26849	*-0.03%*	26835	*-0.08%*	26844	*-0.05%*
Response latency (ms)									
localhost	4.14	52.99	*1179%*	5.14	*24.2%*	89.82	*2068%*	5.16	*24.7%*
gigabit	31.54	53.19	*68.6%*	31.50	*-0.11%*	31.27	*-0.84%*	31.20	*-1.07%*

Connection	Unprotected	protected with MVX with SOCKET_READ_WRITE security policy							
		No IP-MON		IP-MON					
				in-kernel broker		seccomp call gate		user-space call gate	
Throughput (requests per second)									
localhost	98236	9716	*-90.1%*	19132	*-80.5%*	9640	*-90.2%*	19106	*-80.6%*
gigabit	26858	9720	*-63.8%*	19136	*-28.75%*	9614	*-64.20%*	19174	*-28.61%*
Response latency (ms)									
localhost	4.14	52.99	*1179%*	22.69	*447.9%*	43.13	*941%*	23.51	*467.7%*
gigabit	31.54	53.19	*68.6%*	22.82	*-27.63%*	55.90	*77.23%*	22.84	*-27.58%*

Table 1: Performance results for nginx. Results for the FULL_SYSCALLS policy on top, those for the SOCKET_READ_WRITE policy below.

our client continuously requests a 4 KB web page over 4 concurrent connections, during 30 seconds. We ran this benchmark five times for both setups, and for all four MVX configurations plus the baseline.

Table 1 reports the average throughput and response latency for all configurations and setups, both in absolute numbers and relative to the unprotected baseline, for the FULL_SYSCALLS and the SOCKET_READ_WRITE security policies, and using disjoint code layouts as a diversification.

5.2.1. Results for the FULL_SYSCALLS security policy

First, we analyse the FULL_SYSCALLS results. In the gigabit setup, only the MVX configuration without IP-MON suffers from a clear performance hit. This confirms the benefit of IP-MON first reported for ReMon [42]. All three IP-MON configurations have no significant cost in terms of throughput or response latency. We actually measured tiny improvements, which are probably due to accidental interactions between the operating system's scheduler and the MVX engine. Most importantly, we conclude that also for this application, our alternative IP-MON designs requiring no kernel patch offer about the same performance as the in-kernel broker design.

In the localhost network setup, the web server is clearly being stress-tested. The throughput of the MVX configurations based on the in-kernel broker and the user-space call gate both decreased by about 19%, while the throughput of the seccomp-based call gate configuration decreased by a far larger 61%. The increases in response latency are roughly in line with the findings for the throughput, although we were somewhat surprised that the increase for the seccomp-based call gate configuration was almost twice as large as that for the configuration without IP-MON. We conjecture that this might be due to some scheduling effect. As this is a completely unrealistic benchmark situation which we only report for the sake of completeness, we have not investigated it further.

The large difference in performance between the seccomp-based call gate and the user-space call gate for the micro-benchmark like localhost setup confirms the conjecture in Section 3.2.2 that the seccomp-based call gate introduces considerable overhead in system-call heavy workloads compared to the in-kernel broker, while the user-space call gate does not.

5.2.2. Results for the SOCKET_READ_WRITE security policy

The results for the second security policy look entirely different. More system calls are now handled by CP-MON, requiring more context switches, and hence a larger performance hit of using MVX. This is clearly visible in both the localhost and gigabit throughput results. Contrary to the throughputs obtained with the FULL_SYSCALLS policy, which were roughly the same for all three IP-MON designs, the throughput achieved with the seccomp-based call gate under the SOCKET_READ_WRITE policy is much lower that than achieved with the other IP-MON designs under that policy. Interestingly, with MVX protection, for each MVX configuration the localhost and gigabit throughputs are almost identical.

The response latency results can shed a light on this. For the gigabit setup the response latencies decrease from about 32ms for the unprotected (single-variant) configuration to about 23ms for two of the three IP-MON configurations. For the localhost setup, by contrast, the latencies increase from about 4ms to also about 23ms. These different impacts are caused by interaction between the MVX-slowed execution and the process scheduler of the operating system, which can unfold in two different scenarios.

First, when new requests arrive at the webserver by the time previous requests have been handled, the polling mechanism of the server will find a new request as soon as its start polling for one, so the webserver will start handling it without being suspended. This scenario unfolds in the localhost scenario for all configurations and setups, as well as in the gigabit scenario when the handling of tasks by the webserver is slow enough. The latter appears to be the case for all MVX configurations in which at least some system calls are being handled by CP-MON, i.e., for the SOCKET_READ_WRITE policy. In these cases, the delay in handling incoming requests is hence dominated by the execution time that the webserver needs to handle them, which does not depend on the type of connection, but does depend on the MVX and IP-MON configuration and on the security policy.

In contrast, when new requests do not arrive by the time previous requests have been handled, the webserver process will be suspended when it is polling, which starts after handling the previous request and then does not immediately find a new request. In that scenario, the delay in handling incoming requests is dominated by the scheduler, i.e.,

by how long it takes to restart a suspended process. This happens for the unprotected webserver in the gigabit setup for both policies, as well as for the IP-MON configurations under the FULL_SYSCALLS policy. In those cases, on our measurement system, the operating scheduler determined the average response latency, rather than the speed with which the monitored webserver can handle individual requests.

The result is that for the SOCKET_READ_WRITE policy, both the latency increases and the latency decreases are caused by the webserver execution being slowed down because of the monitoring. In the gigabit, this slowdown results in less process suspensions, and hence in the latency decrease.

Even though nginx has been used previously as a benchmark in a number of MVX evaluations, including of ReMon and follow-up research [42, 41, 39, 1], we are the first to report a performance analysis at this level of detail, and hence the first to discuss how the complex the interplay can be between operating system features and multi-variant execution techniques for securing applications, and hence how difficult it can be to predict the performance impact of MVX solutions on real-world applications.

Overall, the numbers still allow us to conclude that the seccomp-based call gate introduces extra overhead compared to ReMon's in-kernel broker, while the user-space call gate does not.

6. Conclusions

This papers proposed two alternative designs for an in-process monitor for use in multi-variant execution engines to achieve secure and efficient application monitoring and replication without requiring a kernel patch. In empirical experiments, the alternative designs proved to be as efficient as ReMon's monitor that required an in-kernel broker. A security analysis concluded that the design is also as secure as ReMon's monitor and in-kernel broker. To achieve the necessary security, modern operating system and hardware functionalities in the form of Seccomp-bpf, Syscall User Dispatch (SUD), and Intel Memory Protection Keys (MPK) were leveraged. Together, they sufficed to implement the necessary isolation and privilege boundaries between the application to be secured and the in-process monitor. Not requiring a kernel patch to achieve the same performance and security will ease adoption of multi-variant execution in practice.

7. Artifact Availability

All scripts used to evaluate the benchmarks for the different MVX configurations are available in the `callgates` branch at `https://github.com/csl-ugent/pirop-scripts`. The callgate-enabled version of the MVX engine is available in the `callgates` branch at `https://github.com/ReMon-MVEE/ReMon`.

8. Funding

The research in this paper was funded by the Cybersecurity Research Program Flanders, and by the Research Foundation - Flanders (FWO) [grant number 3G031320].

References

[1] Abrath, B., Coppens, B., De Sutter, B.: MVX-based mitigation of position-independent code reuse. Computers & Security (2025), accepted for publication

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

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

[4] Cavallaro, L.: Comprehensive Memory Error Protection via Diversity and Taint-Tracking. Ph.D. thesis, Universita Degli Studi Di Milano, PhD dissertation (2007)

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

[6] Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: Proceedings of the USENIX Security Symposium (2005)

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

[8] Corbet, J.: Intel Memory Protection Keys. `https://lwn.net/Articles/643797/` (2015)

[9] Standard Performance Evaluation Corporation: SPEC CPU 2017 benchmark, `https://www.spec.org/cpu2017/`

[10] Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., Hiser, J.: N-variant systems: A secretless framework for security through diversity. In: USENIX Security Symposium (2006)

[11] documentation, T.L.K.: Memory protection keys, `https://docs.kernel.org/core-api/protection-keys.html`

[12] documentation, T.L.K.: Seccomp bpf (secure computing with filters), `https://docs.kernel.org/userspace-api/seccomp_filter.html`

[13] documentation, T.L.K.: Syscall user dispatch, `https://docs.kernel.org/admin-guide/syscall-user-dispatch.html`

[14] Duy, K.D., Cho, K., Noh, T., Lee, H.: Capacity: Cryptographically-enforced in-process capabilities for modern ARM architectures. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 874–888. ACM (2023). https://doi.org/10.1145/3576915.3623079

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

[16] Hosek, P., Cadar, C.: Safe software updates via multi-version execution. In: Proceedings of the International Conference on Software Engineering (ICSE) (2013)

[17] Hosek, P., Cadar, C.: 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)

[18] Hu, H., Chua, Z.L., Adrian, S., Saxena, P., Liang, Z.: Automatic generation of data-oriented exploits. In: Proceedings of the USENIX Security Symposium (2015)

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

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

[21] Kim, D., Kwon, Y., Sumner, W.N., Zhang, X., Xu, D.: 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)

[22] Kim, T., Zeldovich, N.: Practical and effective sandboxing for non-root users. In: 2013 USENIX Annual Technical Conference (USENIX ATC 13). pp. 139–144 (2013)

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

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

[25] Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: Sok: Automated software diversity. In: Proceedings of the IEEE Symposium on Security and Privacy (2014)

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

[27] Maurer, M., Brumley, D.: TACHYON: Tandem execution for efficient live patch testing. In: Proceedings of the USENIX Security Symposium (2012)

[28] Miller, M.: Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. In: BlueHat IL (2019)

[29] Pina, L., Andronidis, A., Hicks, M., Cadar, C.: 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)

[30] Pina, L., Grumberg, D., Andronidis, A., Cadar, C.: A {DSL} approach to reconcile equivalent divergent program executions. In: Proceedings of the USENIX Annual Technical Conference (ATC). pp. 417–429 (2017)

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

[32] Salamat, B., Jackson, T., Gal, A., Franz, M.: 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)

[33] Salamat, B., Jackson, T., Wagner, G., Wimmer, C., Franz, M.: Runtime defense against code injection attacks using replicated execution. IEEE Transactions on Dependable and Secure Computing **8**(4), 588–601 (2011)

[34] Shacham, H.: 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)

[35] Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.: 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)

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

[37] Tarkhani, Z., Madhavapeddy, A.: μtiles: Efficient intra-process privilege enforcement of memory regions. CoRR **abs/2004.04846** (2020), `https://arxiv.org/abs/2004.04846`

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

[39] Vinck, J., Abrath, B., Coppens, B., Voulimeneas, A., De Sutter, B., Volckaert, S.: Sharing is caring: Secure and efficient shared memory support for mvees. In: Proceedings of the Seventeenth European Conference on Computer Systems. p. 99–116. EuroSys '22 (2022). https://doi.org/10.1145/3492321.3519558

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

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

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

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

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

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

[46] Voulimeneas, A., Vinck, J., Mechelinck, R., Volckaert, S.: You shall not (by) pass!: practical, secure, and fast PKU-based sandboxing. In: Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys). pp. 266–282 (2022)

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

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

[49] wrk: wrk - a http benchmarking tool, `https://github.com/wg/wrk`

[50] Xu, J., Xie, M., Wu, C., Zhang, Y., Li, Q., Huang, X., Lai, Y., Kang, Y., Wang, W., Wei, Q., Wang, Z.: PANIC: pan-assisted intra-process memory isolation on ARM. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 919–933. ACM (2023). https://doi.org/10.1145/3576915.3623206

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

[52] Yang, F., Im, B., Huang, W., Kaoudis, K., Vahldiek-Oberwagner, A., Tsai, C., Dautenhahn, N.: Endokernel: A thread safe monitor for lightweight subprocess isolation. In: 33rd USENIX Security Symposium (2024)

[53] Yang, F., Vahldiek-Oberwagner, A., Tsai, C.C., Kaoudis, K., Dautenhahn, N.: Making 'syscall' a privilege not a right (2024), `https://arxiv.org/abs/2406.07429`

[54] Yasukata, K., Tazaki, H., Aublin, P., Ishiguro, K.: zpoline: a system call hook mechanism based on binary rewriting. In: Proceedings of the 2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023. pp. 293–300. USENIX Association (2023), `https://www.usenix.org/conference/atc23/presentation/yasukata`

[55] Yeoh, S., Wang, X., Jang, J., Ravindran, B.: smvx: Multi-variant execution on selected code paths. In: Proceedings of the 25th International Middleware Conference, MIDDLEWARE 2024, Hong Kong, SAR, China, December 2-6, 2024. pp. 62–73. ACM (2024). https://doi.org/10.1145/3652892.3654794, https://doi.org/10.1145/3652892.3654794

[56] Österlund, S., Koning, K., Olivier, P., Barbalace, A., Bos, H., Giuffrida, C.: 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)