2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

Secure and Efficient Multi-variant Execution Using Hardware-assisted Process Virtualization

Koen Koning
Vrije Universiteit Amsterdam
koen.koning@vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

***Abstract*—Memory error exploits rank among the most serious security threats. Of the plethora of memory error containment solutions proposed over the years, most have proven to be too weak in practice. Multi-Variant eXecution (MVX) solutions can potentially detect arbitrary memory error exploits via divergent behavior observed in diversified program variants running in parallel. However, none have found practical applicability in security due to their non-trivial performance limitations.**

In this paper, we present *MvArmor*, an MVX system that uses hardware-assisted process virtualization to monitor variants for divergent behavior in an efficient yet secure way. To provide comprehensive protection against memory error exploits, *MvArmor* relies on a new MVX-aware variant generation strategy. The system supports user-configurable security policies to tune the performance-security trade-off. Our analysis shows that *MvArmor* can counter many classes of modern attacks at the cost of modest performance overhead, even with conservative detection policies.

I. Introduction

For more than a quarter of a century and despite a plethora of proposed solutions, memory errors in C and C++ programs still rank among the most serious security concerns today [1], [2]. Even an unsophisticated memory error exploit like Heartbleed can easily compromise the private data of countless users worldwide with serious consequences [3].

Modern operating systems deploy several measures to protect against memory error exploits, but all of them can be circumvented with varying amounts of effort. For example, widely deployed security defenses such as data execution prevention (DEP) [4], address space layout randomization (ASLR) [5], and stack canaries [6] can all be bypassed by modern code-reuse attacks [7], [8]. Stronger security defenses proposed by the research community, either require recompilation of the program and all shared libraries [9], [10], [11], [12] (limiting deployability), or protect only against a subset of all possible memory attacks (limiting security). As an example, popular control-flow integrity (CFI) solutions that protect against control-flow diversion attacks [13], [14], [15], [16], [17] are ineffective against data-only attacks (such as Heartbleed) and possibly even against control-flow diversion attacks that piggyback on legal control flows in the program [18], [19], [20], [21].

The need for defenses that protect against arbitrary attacks has led to a scramble for more comprehensive solutions—most notably Multi-Variant eXecution (MVX)[1] [22], [23]. MVX systems, first proposed by Cox et al. [23] in 2006, run two or more memory-diversified but semantically equivalent software variants in parallel and detect memory attacks from semantically divergent behavior. These variants run on the same machine (utilizing many-core CPUs) and synchronize at the system call (syscall) level. While such systems have been around for nearly a decade, the run-time performance of traditional MVX implementations [24], [25], [26], [27] is so poor—due to their costly syscall monitoring mechanisms—to make them unusable in practice. Also, the limited variant generation strategies in existing solutions often do not offer adequate protection against more sophisticated memory attacks. Recent MVX efforts have therefore focused either on generating better variants so as to detect (some) modern attacks but with no improvement in performance [26], [24], or on improving the performance by efficient in-process implementations which are, unfortunately, not suitable for security enforcement purposes and target reliability instead [22].

In this paper, we propose *MvArmor*, an MVX system which relies on a new secure and efficient multi-variant design to counter arbitrary memory error exploits. Our design leverages hardware-assisted process virtualization to place the application-level MVX monitor directly in the syscall path of each of the running variants. This approach is efficient, as it does not incur the frequent context switches from/to external monitoring processes required by traditional process tracing-based MVX implementations [24], [25], [26]. Furthermore, given that the process virtualization layer can grant the MVX monitor access to privileged CPU features [28], our design is particularly amenable to optimizations [29], [30]. At the same time, this approach is secure, as it relies on hardware-enforced protection rings to completely isolate the execution of the MVX monitor by construction—unlike prior in-process implementations [22]—protecting it from known and unknown attacks. Furthermore, unlike prior implementations running entirely in the kernel [23], our design separates the application-level MVX monitor from the rest of the system, limiting the trusted computing base (TCB).

To counter arbitrary attacks effectively, we complement our MVX design with a new MVX-aware variant generation strategy, which seeks to provide strong security and performance guarantees with no manual effort. Our strategy relies on per-variant allocator abstractions to carefully and efficiently control the memory layout across the running variants. This strategy provides deterministic security guarantees against arbitrary memory error exploits when possible—or strong probabilistic guarantees otherwise. Finally, *MvArmor* supports flexible security policies tailored to different classes of modern attacks (e.g.,

[1]Also known as N-variant or dual execution and closely related to N-version execution.

arbitrary code execution or information disclosure), allowing users to tune the performance-security trade-off according to their needs.

Summarizing, our contributions are:

- We propose an MVX design based on hardware-assisted process virtualization. Our design efficiently separates the execution of the MVX monitor from both the running variants and the underlying kernel, providing a superior performance and security design point compared to prior efforts.
- We propose a novel variant generation strategy based on MVX-aware allocator abstractions. Our strategy is efficient and, when used in combination within our MVX design, provides strong security guarantees against both traditional and modern memory error exploits.
- We present *MvArmor*, a secure and efficient MVX system. *MvArmor* implements our design on top of Dune [28] to protect commodity Linux programs and offers flexible security policies to encourage deployment. We evaluate *MvArmor* with standard benchmarks and popular real-world server programs and show that *MvArmor* provides a powerful defense against arbitrary memory attacks with much better performance than any existing security-related MVX solution (9% overhead on SPEC CINT2006 and just 55% on average for server applications even with the most conservative security policy).

II. Background

Every MVX system contains two major components: a monitor which runs and synchronizes the variants and a variant generation strategy. Both have a strong impact on the security and performance of the overall system and have been the focus of extensive research in the past decade.

A. Monitor

The MVX monitor is responsible for comparing and synchronizing the execution of the running process variants. These variants all run on the same system, and ideally each have a numbers of cores dedicated to them—we assume that a number of cores can be explicitly dedicated to particularly security-sensitive applications in modern many-core architectures. The monitor itself might consist of several processes, for instance one per variant, communicating via shared memory. The entire MVX system (including the monitor) is designed to be application- and user-transparent. For example, in the case of a web server running under MVX, the user's request will get distributed to all variants. The monitor will also combine the responses of all web server variants and give the user the illusion she is directly talking to a single web server instance. Furthermore, whenever these responses (or other operations) are not equivalent across variants, the monitor can immediately detect an attack attempt (as normal operations should never trigger divergent behavior) and stop the variants before the attacker could do any harm. In general, MVX does not lead to more filesystem and socket I/O, as the monitor effectively executes all syscalls, not every variant. On the other hand, all variants will have to execute all instructions and memory

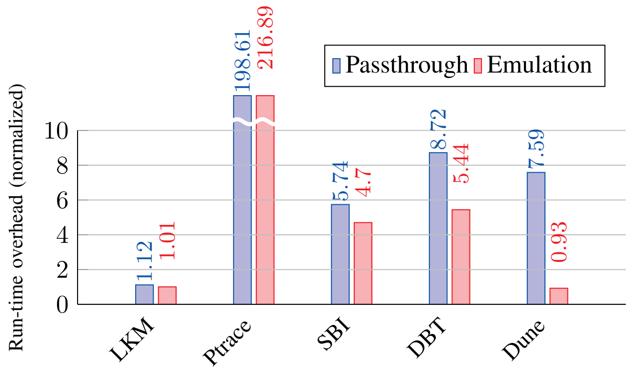

Fig. 1. Overhead induced by several syscall interposition strategies (passthrough and emulation mode) for a microbenchmark repeatedly issuing `getpid` syscalls.

reads/writes by themselves, leading to more overall CPU usage and potential memory bandwidth issues.

In most cases, syscalls are used as synchronization points, as they are generally the primary way for each process to interact with the environment (e.g., file system operations or socket operations). A monitor operating at the syscall level can capture and control external behavior while still allowing for individual variants to exhibit different internal behavior.

A monitor must be able to intercept syscalls and their arguments to compare process behavior across variants. It must also be able to rewrite arguments, block syscalls, and modify the return value (or memory) to ensure uniform and side effect-free syscall handling across all the variants. In general, the monitor needs to ensure all the variants are exposed to the same environment view and information (e.g., PIDs) to avoid unintentionally divergent behavior.

Several strategies have been used in prior MVX systems to intercept syscalls, but all of them suffer from important performance and/or security limitations. To gather insights into these limitations, we evaluated the run-time overhead induced by existing syscall interposition strategies for a simple microbenchmark repeatedly issuing `getpid` syscalls. Figure 1 presents our results in both *passthrough* (i.e., forwarding the original syscall to the underlying OS kernel) and *emulation* (i.e., immediately returning the result to the application) mode.

As shown in the figure, an MVX monitor based on a loadable kernel module (LKM) implements by far the most efficient syscall interposition strategy in both modes of operation, as it does not introduce additional context switches and can directly access the process state. The problem with this strategy, adopted by early MVX systems [23], is that the monitor runs entirely in the kernel. This results in a substantial increase of the trusted computing base (TCB) and poor deployability: a single bug in the monitor could affect the entire system, and the internal kernel API is very volatile.

At the opposite side of the spectrum lie traditional `ptrace`-based MVX implementations [24], [31], [32], [25], [26], which rely on the UNIX process tracing API to implement a deployable but highly inefficient syscall interposition strategy. This strategy introduces multiple context switches between the monitor and the traced process per syscall, resulting in by far the highest overheads (up to ∼217 times) for our microbenchmark. On 64-bit systems a monitor using `ptrace` cannot block syscalls, making emulation even more expensive than passthrough. Syscall monitoring using `ptrace` is also susceptible to TOCTOU attacks [33], which are hard to resolve due to the large latency between operations and limited access between the monitor's and the application's address spaces.

More recent MVX monitor implementations rely on static binary instrumentation (SBI) to rewrite the binary and replace any syscall instruction (e.g., `int $0x80` or `syscall`) with a call into the monitor. As shown in Figure 1, this syscall interposition strategy is much more efficient (up to ∼5 times overhead, based on the Dyninst SBI framework [34]), as it requires no context switches and even no mode switches in emulation mode. Unfortunately, this strategy is not generally suitable for security applications, as an attacker can tamper with the in-process monitor state or simply run uninstrumented unaligned syscall instructions (not part of the normal instruction stream, as x86 does not enforce alignment) to bypass syscall interposition and evade detection altogether. An implementation based on dynamic binary translation (DBT), in turn, would be able to instrument both aligned and unaligned instructions and solve the latter problem, at the cost of slightly higher syscall interposition overhead (up to ∼9 times overhead in our experiment, based on the DynamoRIO DBT framework [35]), but also a non-trivial impact during syscall-free execution. To fully address the former problem, a DBT-based solution needs to deploy additional instrumentation [16] (e.g., software-based fault isolation [36]), but this would also further increase the overhead during syscall-free execution.

MvArmor, instead, relies on hardware-level process virtualization to implement syscall interposition. For this purpose, we use Dune, which virtualizes regular Linux processes and places them in their own (hardware-supported) virtual environment. As shown in Figure 1, this strategy is very efficient compared to other techniques (up to ∼7 times overhead on `getpid()`) and meets all our security demands: small systemwide TCB, fully isolated monitor with no in-process state, non-bypassable (trap-based) syscall interposition mechanism. Furthermore, the excellent performance in emulation mode and the ability to access privileged CPU features provide interesting opportunities for libOS-style optimizations [29], [30].

B. Variant generation

The variant generation strategy has a strong impact on the classes of attacks addressed by the resulting MVX system. Ideally, any attack should eventually result in divergent behavior among variants. For instance, relying on ASLR for variant generation makes it unlikely that two or more variants share code pages at the same addresses, making code-reuse attacks such as ROP [37] more difficult. By extending this strategy to use MVX-aware (i.e. non-overlapping) address spaces, traditional code-reuse attacks can be fully prevented. This is because no code pointer can ever be valid in more than one variant, and thus will cause a fault while dereferencing it in all but one of the variants [23], [25], [27]. While this approach will also stop arbitrary memory read and write attacks that rely on absolute memory object locations (e.g., data pointer overwrites), it cannot stop attacks that only rely on the relative distance between memory objects. For instance, stack- or heap-based buffer overreads that disclose private data (e.g., cryptographic keys) [3] and overflows that corrupt sensitive non-pointer data (e.g., UIDs) [38] will still work reliably, as these attacks only use relative memory accesses.

Alternative variant generation strategies include reversing the direction of the stack [24] and randomizing the instruction set [23]. These strategies add little additional security (e.g., addressing only stack-based and code injection attacks, respectively) and often introduce non-trivial overhead by themselves.

MvArmor, in contrast, relies on a new MVX-aware variant generation strategy, which seeks to minimize the run-time performance impact while providing strong security guarantees against arbitrary classes of attacks that rely on both absolute and relative accesses in memory. Security policies allow the user to choose between increasing levels of protection (both probabilistic and deterministic), at the cost of a larger performance overhead.

Finally, the advantage of using an MVX system over other approaches with the same goals is that it can simultaneously protect against multiple of these types of attacks with less overhead (given enough spare CPU cores and resources), and does not generally require access to the source or recompilation of system libraries [10], [9].

III. Threat model

We assume a strong threat model where an attacker can interact with the target program repeatedly, exploiting vulnerabilities to read or write arbitrary data from/to memory. In particular, we assume an attacker can rely on both *relative* (e.g., buffer overread/overflow and partial pointer overwrites) and *absolute* arbitrary memory read/write primitives (e.g., pointer overwrites). We also assume both *spatial* (e.g., buffer overflows) and *temporal* (e.g., use-after-free) memory attacks [39]. Based on these primitives, we assume an attacker may pursue any of the following goals (in line with the characteristics of modern attacks [18], [40]):

- **Arbitrary code execution**: An attacker could execute arbitrary code, for example issuing an `execve` syscall using ROP [37] or other code-reuse techniques [41].
- **Information disclosure**: An attacker could leak sensitive data from the target program, for example cryptographic keys as in the case of Heartbleed [3].
- **Information tampering**: An attacker could tamper with sensitive data, for example UIDs to escalate privileges, or mount other non-control-data attacks [38], [40].

IV. Overview

Figure 2 shows the main components of *MvArmor*, which all run inside a virtualized environment [28]. Additionally, *MvArmor* can provide the same functionality for other syscall interception methods by simply replacing the syscall frontend and backend components. At startup, the *variant generator* (Sec. V-A) spawns an application instance for every variant, using an MVX-aware variant generation strategy in order to protect applications from all the previously mentioned attack vectors. The *security manager* (Sec. V-B), in turn, generates security policies, providing a trade-off between security and performance for the rest of the components. These security policies can be defined by the user and depend on the classes of attacks considered.

When the application executes a syscall, it will trap into the *syscall frontend* (Sec. V-C). This component is the entry point into both the monitor and the "kernel" (ring 0 code) of the virtual environment. The syscall frontend also manages all accesses to application state, such as its address space.

The frontend forwards all syscall events to the *variant manager* (Sec. V-D), which is responsible for synchronizing all variants and enforcing the security policies. The variant managers communicate with each other using a ring buffer similar to that proposed by Hosek and Cadar [22]. Specifically, one of the variants (the leader) performs all of the actual syscalls and sends the corresponding events to the other variants (followers), who consume the events in their own time. The followers only execute a small set of these syscalls as well (e.g., memory management) as most I/O should happen only once (e.g., sending data over a socket). Unlike traditional MVX systems, the variants can run asynchronously most of the time, removing the great performance bottleneck of running variants in lockstep. However, unbridled asynchronicity is not safe. It would allow one variant to achieve arbitrary code execution with calls like `exec`, or information disclosure with calls like `write`. Instead, we use the known distinction between security-sensitive and non-security-sensitive syscalls [42], [43] and enforce selective lock-step execution, where the set of sensitive calls varies depending on the security policy.

The variant manager is also responsible for deciding when a syscall should really be executed (leader) or when the results should simply be copied (followers). When the variant manager decides to execute a syscall, it sends it to the *syscall backend* (Sec. V-E). In a naive implementation, the backend would simply forward all syscalls to the real kernel. However, each syscall in Dune requires a costly VM exit. To reduce these costs, we implemented a set of (memory management and `getpid`-like) syscalls directly in our monitor. Further libOS-style optimizations are also possible—for instance, by using a userspace network stack such as IX [29], or by batching syscalls [44].

The variant manager uses the *namespace manager* (Sec. V-F) to ensure all information available to variants is the same (including PIDs, file descriptors, and timing information), and finally the *detector* (Sec. V-G) to semantically compare the execution of the variants for divergence.

V. *MvArmor*: FAST AND SECURE MVX

We now describe each of *MvArmor*'s components in detail.

A. *Variant generator*

A fundamental question in MVX is to what extent the variants should differ. Unconstrained variation makes it impossible to detect attacks from divergence, as *everything* may be different. Conversely, insufficient variation is also undesirable, as there may not be *any* divergence for an attack. Fortunately, for memory errors the straightforward solution is to vary the address space layout and keep everything else the same, as these differences should normally not affect program execution

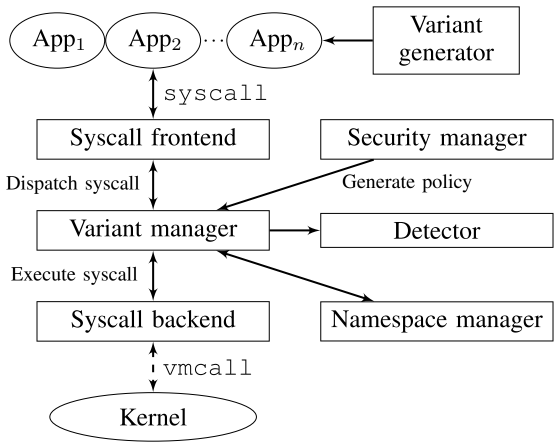

Fig. 2. Overview of all MVX design components.

but will make a difference in the case of malicious memory actions. In this section, we identify several techniques that offer strong protection and detection against different classes of memory error exploits and we detail the corresponding implementation strategy in our current *MvArmor* prototype. For our analysis, we assume the now common PIE binary organization [45], but our design can, in principle, also handle non-PIE binaries by marking static program segments as non-relocatable and gracefully reduce security guarantees.

First of all, by using *non-overlapping address spaces* across variants (pioneered by [23]), any *absolute spatial* attack (i.e., attack relying on absolute code/data addresses) is already rendered ineffective. By ensuring that memory pages do not overlap across variants, a pointer can only be valid in at most one variant at a time and will thus *deterministically* crash all others. This already stops common code-reuse attacks such as ROP [37] and information disclosure attacks such as JIT-ROP [46], as they rely on the absolute position of memory pages. In fact, even any other attempts (e.g., buffer overreads) to disclose code or data pointers will also cause divergence, as different values will be leaked to the attacker at the syscall level (e.g., over a socket) by construction. To enforce non-overlapping address spaces across variants, we randomize each variant using ASLR and then constrain ASLR not to reuse address ranges across variants. Since our MVX system resides in ring 0 of the virtualized environment, it has full control over the page tables, making ASLR modifications simple. *MvArmor* implements this technique for all the memory regions (i.e., code, data, stack, etc.) in each variant.

To also *deterministically* stop *relative spatial* attacks (i.e., attacks relying on relative code/data addresses), our variant generation strategy must be able to provide strong guarantees against buffer overflows/underflows and partial pointer overwrites. We observe that, for this purpose, our strategy must simply ensure that *offsets* between memory objects are non-overlapping. For example, if the size between objects on, say, the heap in a follower is as large as the entire (normal and compact) heap in the leader, any offset added to a pointer can only be valid in one of these variants. In other words, this design ensures *non-overlapping offset spaces* across variants, rendering all the relative spatial attacks ineffective. *MvArmor* implements this novel technique for all the heap objects, by

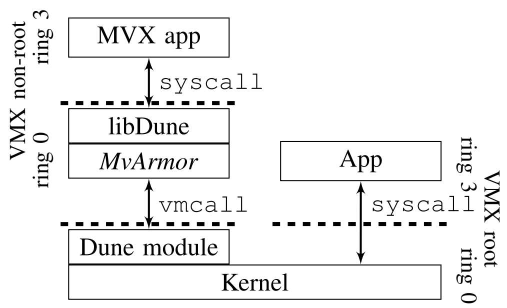

Fig. 3. Control flow of syscalls with and without Dune.

using the standard "*compact*" allocator in the leader and a custom "*sparse*" allocator in the followers. Extending such guarantees to all the other memory objects is, in principle, possible, but source-level information is generally necessary to accurately decouple stack [47] and data [48] objects—although binary-level approximations are at times possible [49].

While the strategies described thus far can provide deterministic protection against all the spatial attacks, they are alone insufficient to stop *temporal attacks* (e.g., use-after-free exploits). Unfortunately, ensuring deterministic protection guarantees against generic temporal attacks is not practical without source-level information [50]. A practical binary-level alternative is to ensure *probabilistic* temporal safety guarantees. In our design, this is done by using different (randomized) memory allocators across variants and, to further limit the attack surface, by approximating type-safe memory reuse [51] at the binary level. *MvArmor* enforces probabilistic temporal safety for all the heap objects, randomizing the standard allocator with random inter-object gaps in the leader. In addition, *MvArmor* overapproximates type-safe memory reuse using per-size memory pools in our custom allocator in the followers (but much less conservative binary-level approximations based on allocation-time backtraces are also possible [51]).

While the implementation of all the proposed protection techniques can introduce significant overhead when all combined together on a single variant, their overhead can, in most cases, be completely masked across variants with our MVX design. In *MvArmor*, followers are faster than the leader, as they do not execute most syscalls and thus waste several cycles waiting for the leader. Our measurements show that, for our baseline *MvArmor* implementation (i.e., without any protection enabled) on (I/O bound) server applications, the followers spend around 4,000 cycles on average per syscall waiting for the leader. Especially when syscalls do not require lockstep behavior, the idle periods leave sufficient time for the followers to spend more time in more expensive allocator abstractions implementing our protection techniques. This strategy provides strong security guarantees while reducing the run-time overhead of the end-to-end solution.

B. Security manager

The security manager generates policies that allow users to make trade-offs between security and performance. Specifically, a policy specifies whether each syscall is considered *non-sensitive* (event-streaming, meaning the leader can execute it without synchronization), or *sensitive* (requiring lockstep execution with other variants).

Security policies specify behavior at the level of the whole system, individual syscalls, or even specific arguments (e.g., "If the execute bit in a permissions flag is set then..."). In *MvArmor*, we propose the following policies for each of the aforementioned classes of attacks (but others are possible):

- *Code execution*: enforce full checks on `execve` and `mprotect`/`mmap` with execute permissions set.
- *Information disclosure*: enforce full checks on I/O syscalls that are able to leak data (e.g., `write`).
- *Comprehensive*: full checks on all syscalls.

In practice, the *Code execution* policy performs as efficiently as a policy where no syscalls are considered sensitive, since the syscalls considered by the *Code execution* policy are rarely executed in most applications.

The *Comprehensive* security policy, in turn, is useful to provide a generic catch-all strategy (and a lower bound on performance) but, given a target threat model, may provide comparable security to more tailored policies such as *Code execution* and *Information disclosure*. The key insight is that such security policies may delay detection of *failed* attacks, but they do deterministically and immediately stop successful attempts for all the attacks considered in the threat model.

C. Syscall frontend

When the application executes a `syscall` instruction, the execution will trap from ring 3 into ring 0—kernel space. By running the application and the monitor in a virtualized environment, all syscalls will trap into the monitor instead of the actual kernel. *MvArmor* is based on Dune [28], which leverages virtualization to provide applications with access to privileged CPU features in a safe way. For this purpose, Dune relies on Intel VT-x extensions to allow a core to temporarily switch from the normal kernel (*VMX root*) into virtualized mode (*VMX non-root*) via the Dune hypervisor. Dune sets up ring 0 code for both *VMX root* and *non-root* mode, as shown in Figure 3. Since our monitor runs in privileged mode, it can also access other features, such as page tables and interrupts. While the same effect could be achieved by a kernel module or by modifying the kernel directly, *MvArmor* completely separates the monitor from the rest of the system, with no systemwide TCB increase.

The syscall frontend receives `syscall` traps from *libDune* and forwards them to the variant manager. In addition, the frontend is responsible for access to the application state, such as reads and writes to its address space.

Not all syscalls incur a trap on modern Linux systems; in every application, the kernel sets up a shared library (the virtual dynamic shared object—i.e., vDSO), which contains code to execute a selection of syscalls without trapping into the kernel. Using Dune, we can still intercept vDSO calls, by mapping our own code containing `syscall` instructions in place of the original vDSO.

D. Variant manager

Upon receiving a syscall event from the frontend, the variant manager synchronizes with the other variants. Every process has a ring buffer it shares with the respective processes in other variants. Upon completion of a syscall, the leader pushes it into the ring buffer together with its arguments and return value. The followers compare their arguments to those of the leader, and either execute the syscall themselves, or use the return value provided by the leader. Specifically, the variant manager has a per-syscall table to determine the appropriate behavior. Certain syscalls should execute only once (e.g., socket-related syscalls), while others should execute in every variant (e.g., memory management calls). For the former, the followers simply copy the return value of the leader.

The ring buffers resemble those in Varan [22] and provide efficient communication without locking. We use atomic operations to update the ring buffer entries and busy-waiting to consume them. As syscalls like `select` and `epoll_wait` may block for a long time, followers stop busy-waiting after some time and go to sleep instead. Doing so is expensive due to the VM exits required for both sleep and wake-up calls. Assuming there is no shortage of cores, a more efficient solution is to sleep using Intel's `monitor`/`mwait` instructions as they incur no VM exit. Because our Dune-based virtualized environment runs in privileged mode, it can trivially use them where normal applications cannot.

As mentioned earlier, the security policy determines whether each syscall should run in lockstep, in which case the leader waits for all followers after pushing the arguments into the ring buffer. The followers then compare the syscalls as usual and then pause while the leader finishes the syscall. Doing so for every syscall (most conservative security policy) has a higher performance impact.

For multi-threaded applications, we force the followers to adhere to the order of syscalls of the leader. This strategy seeks to prevent divergent behavior due to non-deterministic scheduling decisions. This loose form of deterministic multithreading (DMT) was shown to be generally sufficient for previous MVX systems [32], [22]. Full DMT semantics could be used if issues do occur (such as divergence because of benign data races), at the cost of larger overhead [52], [53], [54].

E. Syscall backend

When a variant needs to execute a syscall, it forwards the call to the syscall backend. Forwarding syscalls to the kernel requires costly VM exits, so the syscall backend tries to execute the syscall locally when possible. This is currently implemented for memory management syscalls, but could be, for example, extended to include userspace network stacks such as IX [29]—which is also based on Dune. Besides eliminating VM exits, userspace network stacks also improve the overall performance of the application.

MVX monitors can be susceptible to time-of-check-to-time-of-use (TOCTOU) attacks, where an attacker modifies the arguments of a syscall in memory from a different thread after the arguments are checked by the monitor but before they are read by the kernel. This works because the arguments passed to the syscalls are usually pointers to buffers or structures in the address space of the application, copied separately by the monitor (for checking) and the kernel (for execution). We solve this by directly passing the pointers to the copied data structures (in the monitor) to the kernel. Because no additional copying is required, this introduces no performance overhead.

F. Namespace manager

The namespace manager ensures the variants do not diverge accidentally by eliminating all variant-specific information. For instance, if variants were to have access to their kernel-assigned PIDs or timing information, they may (directly or indirectly) use such data in a conditional, leading to divergent behavior. The namespace manager therefore assigns virtual PIDs and TIDs to every process and thread using a hierarchical structure: when a variant creates threads in quick succession, these must get the same virtual TID in all variants, regardless of the order they actually appear on the system or the thread that executed its `clone` operation earlier.

We similarly virtualize file descriptors, as only the leader has access to all of them. For instance, followers do not have access to sockets or files opened as writable. Since the kernel assigns file descriptors in an incremental fashion per process and the followers open fewer files (e.g., read-only files) than the leader, these numbers start to diverge. The namespace manager therefore maintains a mapping of virtualized file descriptors to real (per-variant) file descriptors. The same holds for epoll-related identifiers, including the user data field. The `epoll_wait` syscall returns user-defined data previously registered when a socket has an I/O event. Since these values can be pointers (that differ per variant), they have to be mapped back to the socket and then to the variant-specific user data that should be returned for that socket.

Timing information should also not differ among variants, as this is often used for logging or seeding the random number generator. Since *MvArmor* has full control over the page tables of the application, it can easily intercept all vDSO syscalls. While not commonly used, we also disable the `rdtsc` instruction so that it traps into the monitor.

We ensure determinism in random number generation by allowing only the leader to open files like `/dev/random`. Pseudorandom number generators are generally seeded with information already virtualized by the namespace manager and require no additional effort to work correctly. We similarly limit access to the `/proc` file system to the leader. Without binary instrumentation, there is no easy way of interposing `rdrand` instructions. By disabling the corresponding bit in the `cpuid` implementation of the hypervisor, most normal applications and libraries will not use it (e.g., OpenSSL). While we have not observed the need to check on this further, we could also configure the virtual environment to trap into the hypervisor when the instruction is executed (via a bit in the control structure of the virtual environment).

G. Detector

Since a syscall can take no more than six arguments, many calls expect pointers to data structures which hold more information (e.g., a buffer or `struct`). For a full comparison between variants, the monitor therefore performs a deep semantic copy and comparison of such arguments [23],

[32]. In *MvArmor*, the detector component performs both of these functions.

H. Implementation

MvArmor consists of a library implementing all the components in our design except for the frontend and backend. We developed two implementations of these components: our high-performance hardware-virtualization approach using the Dune sandbox and a `ptrace` implementation for development and debugging. These implementations call our shared library for every syscall and expose several functions such as how to access the monitored applications' address space and how to allocate memory across variants. The library itself consists of around 5,000 lines of C code, whereas the frontends and backends include around 500 lines of C code each.

The Dune sandbox, which is used to implement the default frontend and backend of *MvArmor*, allows for arbitrary applications to run in Dune. The sandbox loads a given binary using its own loader. It also implements bounds checking on any pointer passed to a syscall to prevent sandboxed applications from accessing ring 0 state such as the sandbox itself, the Dune library, or the monitor. We slightly modified both Dune and the sandbox to meet our requirements for the monitor, such as security fixes and more callbacks.

To implement the protection techniques discussed in Sec. V-A, we used a modified version of `libumem`[2], a Linux userspace port of the Solaris slab allocator [55], [56]. This implementation served as a basis for our custom "*sparse*" allocator. In particular, by limiting the number of objects allowed (i.e., 1 object) per slab and adding padding (i.e., the leader's maximum heap size) to every slab, we enforce *non-overlapping offset spaces* between leader and followers. By preserving the natural per-size pooling architecture of `libumem`, we overapproximate type-safe memory reuse. Furthermore, since we want to retain the standard (randomized) allocator in the leader (to preserve security, but also performance guarantees in the slower leader), we assume both the standard and our custom allocator as trusted to prevent the monitor from detecting divergence caused by the different allocators (e.g., different syscalls to map memory). Note that while our custom allocator reduces ASLR entropy, this is irrelevant as our MVX security guarantees are stronger, and not ASLR-dependent.

VI. LIMITATIONS

At the time of writing, our *MvArmor* prototype has the following limitations:

- *MvArmor*'s custom allocator is subject to Dune's restrictions on the maximum per-process virtual memory size, which currently requires relaxing the size restrictions on inter-slab padding (and thus security) in memory-intensive applications.
- While *MvArmor* can protect generic heap objects, it cannot decouple intra-struct buffers or chunks managed by custom memory allocators within each object without source-level information, a limitation fundamental to all the binary-level heap hardening solutions [51].
- While *MvArmor*'s MVX library supports threading similar to recent MVX solutions [22], it cannot currently run multi-threaded applications. Extending our current *MvArmor* prototype to support arbitrary multi-threaded applications faces two challenges: (i) supporting thread safety in Dune (currently thread-unsafe), and, when benign data races are present (i.e., threads synchronizing without syscalls such as `futex`), (ii) preserving correct MVX semantics with a more strict form of DMT.

[2]https://github.com/gburd/libumem

VII. EVALUATION

We evaluated *MvArmor* on a workstation with an Intel i7-3770 quadcore CPU clocked at 3.4 GHz and 16 GB of RAM. We disabled hyperthreading to eliminate (large) fluctuations in our test results. We ran all our experiments on a Debian 8.0 system, running a Linux kernel 3.2 (x86_64).

For our evaluation, we considered a number of popular server programs, which are heavily exposed to remote attacks (and thus would greatly benefit from the security guarantees provided by *MvArmor*) and have also been extensively benchmarked in prior work. In particular, we selected nginx (v0.8.54), lighttpd (v1.4.28), bind (v9.9.3), and beanstalkd (v1.10) for our experiments. We benchmarked bind, a popular name server, using the queryperf benchmark issuing 500,000 requests with 20 (default) threads. We benchmarked nginx and lighttpd, both high-performance web servers, using the wrk benchmark issuing 10 seconds worth of requests for a 4 KB page over 10 concurrent connections. We benchmarked beanstalkd, a work queue, with the beanstalkd benchmark issuing 100,000 push operations per worker over 10 concurrent connections and 256 bytes of data. To directly compare against Varan [22] (by far the fastest, but not security-oriented, state-of-the-art MVX solution), we adopted the same benchmark configurations (wrk and beanstalkd) considered in [22]—only increasing the number of push operations in the beanstalkd benchmark by a factor of 10 to ensure a sufficient benchmark duration (i.e., 10-20 seconds).

We also evaluated *MvArmor* on microbenchmarks and on the SPEC CPU2006 benchmark suite, focusing our experiments on the CINT2006 benchmarks to reflect the configuration considered in [22] and provide comparative results. We ran all our experiments 11 times and report the median (with small standard deviation across runs). We report results for our default *MvArmor* configuration using a 10-element ring buffer (allowing the leader to execute 10 syscalls ahead of followers), but we observed similar results when moderately increasing/decreasing the ring buffer size. Unless otherwise noted, our experiments use the variant generation strategy from Sec. V: the leader uses the default (randomized) allocator, whereas each follower uses the modified `libumem` allocator.

A. Server Performance

To evaluate *MvArmor*'s performance on our server programs, we first attempted to reproduce the over-the-network configuration described in [22], placing the client on a dedicated machine on the same rack as the server machine, with the 2 machines connected by a 1 Gbit/s ethernet link. In our setup, this configuration was insufficient to effectively saturate the server, reporting only marginal performance impact across

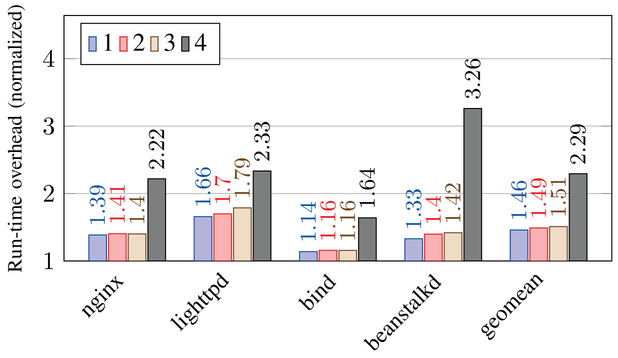

Fig. 4. Overhead using the *Code execution* security policy for increasing number of variants.

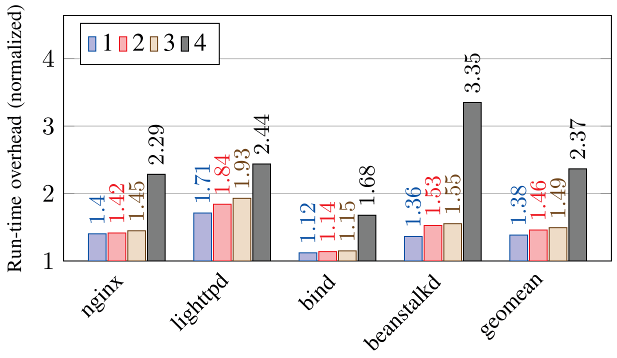

Fig. 6. Overhead using the *Information disclosure* security policy for increasing number of variants.

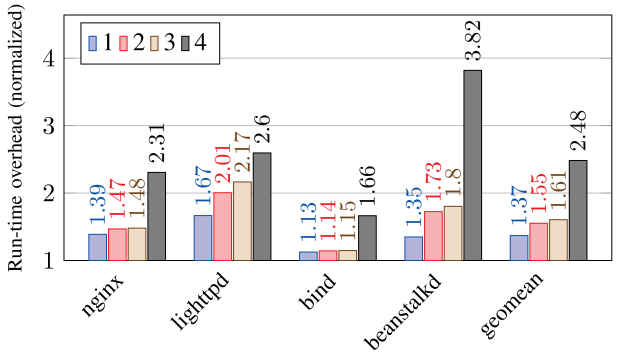

Fig. 5. Overhead using the *Comprehensive* security policy for increasing number of variants.

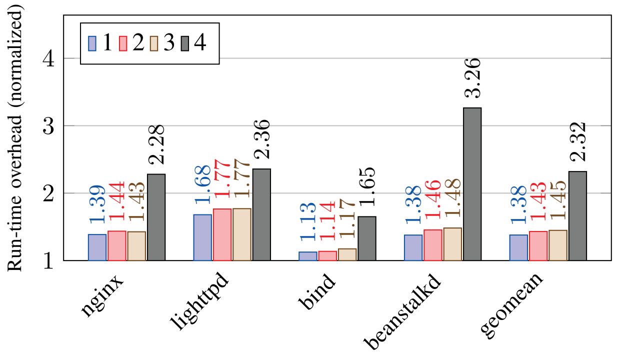

Fig. 7. Overhead using the *Information disclosure* security policy for increasing number of variants with our variant generation strategy **disabled**.

all our programs. To fully saturate all our server programs, we then reran our benchmarks through the loopback interface. We note that this strategy resulted in sound but also more pessimistic performance results. As an example, the original Dune paper reports ∼1% overhead on lighttpd for an over-the-network configuration [28]. Reproducing the exact same experiment in our setup (which, in contrast, relies on the loopback interface) resulted in much higher impact (∼12%). This is caused by less networking overhead, making the impact of our system more visible.

We evaluated *MvArmor*'s performance across all the security policies supported. Figure 4 shows the results for the *Code execution* security policy for an increasing number of variants (1 through 4, where 2 variants means 1 leader and 1 follower). Since the server applications do not normally execute any syscalls that fall under the *Code execution* policy, the results are identical to a policy where no syscalls are considered sensitive. When disabling our variant generation strategy (isolating the MVX synchronization overhead) for both policies, we observed no differences in the results. This shows that the followers are indeed able to keep up with the leader despite the less efficient secure allocator, hence our variant generation strategy has essentially no impact. Note that security policies have no effect on scenarios with only one variant, as there is no synchronization in such scenarios.

Figure 6 reports results for a slightly more conservative policy (*Information disclosure*). As our server applications make heavy use of write syscalls, which run in lockstep now (forcing idle time in the followers to be spent waiting for the leader rather then performing extra operations), the overhead of the slower followers (and of our variant generation strategy) cannot be completely masked in this case. For comparison purposes, Figure 7 reports results for the same policy, but with our variant generation strategy disabled. Finally, Figure 5 reports results for our most conservative security policy possible (*Comprehensive*, generally overly conservative, but useful to provide worst-case results). In this configuration, the impact of our variant generation strategy (and custom allocator in the followers) is more noticeable given that all the syscalls run in lockstep (e.g., 55.2% vs. 49.1% for two variants, geometric mean—or geomean).

As shown in the figures, programs with a lower number of "copy-heavy" syscalls such as nginx and lighttpd scale less efficiently with the number of variants and are also more affected by the increasingly lockstep-like behavior enforced by more conservative security policies (e.g., full lockstep for *Comprehensive*). Beanstalkd, on the other hand, issues relatively few syscalls overall and the reported overheads are mostly due to the copying costs for Beanstalkd's large buffers. This results in Beanstalkd performance being non-trivially impacted by our syscall interposition strategy, but scaling well with the number of variants and with more conservative security policies. We observed similar behavior for bind, which only issues 2 very "copy-heavy" syscalls per request (i.e., `recvmsg` and `sendmsg`).

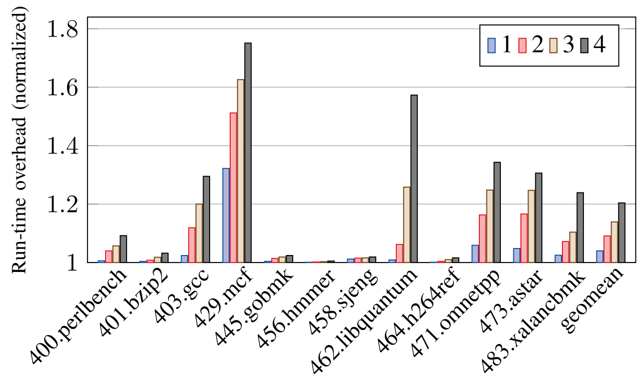

Fig. 8. Overhead for the SPEC CINT2006 benchmarks for an increasing number of variants.

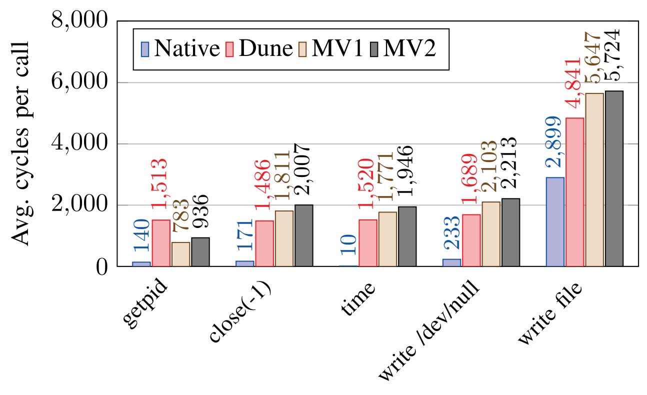

Fig. 9. Average number cycles for various syscalls. Dune (using the sandbox app) always runs in passthrough mode. *MvArmor* runs with 1 (MV1) or 2 (MV2) parallel variants using the *Code execution* security policy.

Overall, our results suggest that *MvArmor* scales well with the number of variants, for example with only a ∼3% average overhead increase (geomean) when moving from a 2-variant to a 3-variant configuration (across different security policies). We observe significant performance drops only when exhausting the number of cores—a 4-variant configuration in our 4-core setup, with a core dedicated to the benchmark program. We also note that a more large-scale scalability analysis, while possibly interesting, is irrelevant in practice. *MvArmor*'s default configuration using 2 variants is sufficient to provide strong security guarantees by construction and also minimizes core utilization to encourage deployment in real-world settings.

We now compare our results against Varan [22]. For a fair comparison, we focus on the *Code execution* security policy, which most closely matches Varan's MVX strategy. *MvArmor*'s performance is very similar to Varan for nginx (41% vs. 37% for 2 variants, respectively), but we did observe somewhat different results for beanstalkd and lighttpd. In particular, for beanstalkd, our results show relatively low and stable overheads for *MvArmor* (∼41%) and increasingly higher overheads for Varan (52% and 57%, for 2 and 3 variants, respectively). Conversely, for lighttpd, Varan reports low and nearly constant overheads (∼14%), while *MvArmor*'s overheads start at 70% for 2 variants.

Overall, *MvArmor*'s overhead results are generally comparable to Varan, although our experiments show that the actual performance of the 2 systems depends on the program considered. We believe our results are very encouraging, given that (i) Varan is the fastest existing MVX implementation, (ii) compared to Varan, *MvArmor* provides much stronger security guarantees even for our least conservative (*Code execution*) security policy, (iii) our loopback-based performance results are pessimistic and could also be improved by operating further libOS-style optimizations enabled by our design.

B. SPEC Performance

To further compare our results against prior solutions, we evaluated the performance impact induced by *MvArmor* on the SPEC CINT2006 benchmarks. While the SPEC benchmarks are CPU-intensive and issue a relatively low number of syscalls (thereby providing optimistic performance results for MVX systems), this experiment still provides useful comparative data points. Figure 8 presents our findings.

As shown in the figure, *MvArmor* yields an average overhead of 9.1% (geomean) for 2 variants and 20.4% for 4 variants across all the benchmarks. A closer inspection revealed that, in most cases, the reported overheads primarily originated by the impact on the memory bandwidth of the system. Benchmarks such as `mcf` and `libquantum` are particularly memory-intensive and tend not to scale well in simultaneous runs on multi-core architectures [57].

Nevertheless, despite the greater impact of TLB misses in memory-intensive benchmarks induced by the use of EPTs in Dune [28], our results are encouraging and, in fact, even yield better performance than Varan, the best MVX performer on SPEC in the literature, reporting an average overhead of 14.2% (geomean) with 2 variants [22] on the same set of benchmarks.

C. Microbenchmark Performance

To carefully pinpoint the sources of overhead introduced by *MvArmor*, we evaluated our solution using a number of microbenchmarks. In particular, we measured the number of cycles required by various syscalls from the perspective of a user program while running under Dune [28] and under our full *MvArmor* solution with 1 or 2 variants (MV1 and MV2, respectively). Figure 9 presents our findings.

Both the `getpid` and `close(-1)` syscalls have a very short duration, with the kernel almost immediately returning to userland. For these simple syscalls, Dune alone adds around 1,300 cycles, accounting for syscall interposition and (mostly) for the `vmcall` to the host. *MvArmor*, in turn, adds around 300 extra cycles on top of Dune. Our microbenchmark results are compatible with those in the original Dune paper [28]. As the `getpid` syscall is currently implemented in *MvArmor*'s backend, it does not require an expensive `vmcall`. This is reflected in the significantly reduced overhead compared to simply running Dune in passthrough mode (783 vs. 1,513 cycles), even with the additional MVX logic in place. This shows libOS-style optimizations are a viable strategy to speedup *MvArmor* in the future.

The `write` syscall has a buffer argument, which first undergoes bounds checking in Dune and then requires copying the buffer in *MvArmor*'s monitor. For a small 5-byte buffer, Dune alone adds around 1,500 cycles, while *MvArmor* adds around 400 (`/dev/null`) and 800 (filesystem) extra cycles. When writing to `/dev/null`, the kernel does not have to wait for I/O, as opposed to the case of writes to the filesystem. Since the overheads added by Dune and *MvArmor* are fairly constant, the overall performance impact quickly becomes insignificant with more lengthy I/O requests—such as those typically issued by server programs.

The `time` syscall, part of the vDSO, can natively be executed without a `syscall` instruction, but Dune remaps the vDSO to force traps into the monitor. While this strategy introduces a non-trivial performance impact (around 1,050 cycles for Dune alone), it also allows *MvArmor* to monitor and alter its return value to ensure consistent variant behavior.

D. Security

To analyze the effectiveness of our variant generation strategy against memory errors, we present an analytical security analysis on different classes of exploits and draw from real-world examples. We note that, since an empirical evaluation of security against arbitrary existing exploits would have trivially detected deviations (and thus attacks) in all cases, we opted for an analytical analysis similar to prior work on randomization-based solutions [48], [39].

First, memory error exploits that rely on absolute addresses are *deterministically* prevented by *MvArmor*'s non-overlapping address spaces across variants, regardless of the particular security policy deployed. These exploits can be used to mount many classes of attacks, ranging from modern code-reuse attacks [37], [41] to information disclosure attacks [8].

To exemplify the security guarantees provided by *MvArmor* for these classes of attacks, we consider an exploit based on a real-world vulnerability (CVE-2004-0488). This vulnerability allows an attacker to mount a stack-based buffer overflow exploit against Apache httpd, corrupting a data pointer with an absolute address and granting the attacker the ability to read arbitrary memory values [8]. While this attack is effective against 1 standalone variant (assuming the attacker can bypass ASLR [46] and disclose the intended absolute memory address), an attacker will not be able to find a single absolute memory address which is, at the same time, valid across 2 variants running in parallel—resulting in at least 1 protection fault and *MvArmor* detecting the attack.

Memory error exploits that rely on relative addresses are also *deterministically* prevented by *MvArmor*'s MVX-aware allocator design deployed in the follower(s). These exploits can be used to mount many classes of attacks, e.g., information disclosure/tampering, and other non-control data attacks [40].

To exemplify the security guarantees provided by *MvArmor* for these classes of attacks, we consider two real-world exploits crafting relative memory read and write primitives to achieve the attacker's goals (respectively). We also speculate on an attacker extending these exploits by using temporal vulnerabilities, to demonstrate how *MvArmor* would *probabilistically* prevent more advanced attacks.

For the former case, we consider an exploit based on the Heartbleed vulnerability in the OpenSSL library (CVE-2014-0160). The exploit overreads a buffer located on the heap to read security-sensitive data from other heap objects and eventually allow the program to leak them over the network. With *MvArmor* deployed, the attacker can only read data from the leader because of the non-overlapping offset spaces. Any attempt to read the object in the follower(s) as well would require reading a size larger than the leader's heap, causing the leader to read past its heap and crash. Even if an attacker were to find, say, a use-after-free vulnerability to leak the same security-sensitive data, *MvArmor* would still probabilistically stop the attack. Since the data is leaked using standard I/O syscalls, *MvArmor*'s *Information disclosure* security policy can immediately identify the divergent behavior of reading probabilistically different data from different objects in the leader and the followers (due to the different and randomized allocators, as well as type-safe reuse in the follower(s) increasing the gap), and thus still detect information disclosure.

For the latter case, we consider an exploit based on another vulnerability in the OpenSSL library (CVE-2014-0195). The vulnerability allows an attacker issuing a long non-initial fragment to overflow a heap-allocated buffer and corrupt adjacent data. The exploit relies on this primitive to corrupt security-sensitive non-control data in other heap objects. With *MvArmor* deployed, the attacker is, again, forced to overflow more data to compensate for the inter-object gaps on the heap in the follower(s) and reach the intended security-sensitive data (e.g., UID) across all the variants. However, any attempt to "spray" this much data would again result in protection faults in at least one of the variants, as described earlier. Similarly, even if an attacker were to find, say, a use-after-free vulnerability to tamper with the same security-sensitive data, any write will, again, likely result in different side effects across variants and probabilistically stop the attack.

Finally, since *MvArmor* captures deviations in external behavior by monitoring differences in security-sensitive syscall patterns, the detection guarantees provided against such attacks improve with the conservativeness of the security policy deployed. Note that when not deploying our most conservative security policy (*Comprehensive*), *MvArmor* may fail to detect some *failed* attack attempts immediately, but will still detect (and disallow) all the behavioral deviations induced by successful attack attempts that affect security policy-defined syscalls.

VIII. Related Work

The idea of using software diversity to improve fault tolerance was first introduced by Avižienis and Chen [58] in the seventies. Their idea of N-version programming had multiple teams of programmers implementing the same software, hoping bugs would be isolated to only one of the versions. This paradigm was only later expanded to security applications [59]. In 2006, Cox et al. [23] introduced the idea of using automatically generated variants, rather than versions, to improve deployability. They also analyzed multiple monitoring strategies, such as syscalls (using a kernel module) and a network proxy, and several variant generation strategies, such as disjoint memory mappings and instruction set randomization. DieHard [27], also published in 2006, proposed a probabilistic memory safety solution including a "replicated mode". While similar in spirit, *MvArmor* provides far better performance and security. Variant synchronization issues, in turn, have been the focus of extensive research ever since [25], [24], [32], [31], [26].

Salamat et al. [24] describe other variant generation strategies in Orchestra, proposing a reversed stack in one variant to prevent stack smashing attacks. Orchestra relies on the `ptrace` API to interpose syscalls, similar to most existing MVX monitors [25], [32], [31], [26]. In 2015, Hosek and Cadar proposed Varan, which relies on static binary instrumentation in order to significantly improve MVX performance. In contrast to *MvArmor*, Varan focuses on software reliability rather than security, similar to other MVX-like systems such as Tachyon [32] and Mx [22].

Varan's event-streaming design shares similarities with *MvArmor*'s variant synchronization strategy, in that they are both based on a ring buffer design inspired by existing high-performance lock-free ring buffers [60], [61]. The key difference is that Varan's event-streaming design is fully asynchronous (other than not isolated from the untrusted program execution) and unable to support the synchronous detection policies employed by *MvArmor*'s design for security. Varan's event-streaming architecture shares, in fact, similarities with record-and-replay systems, in which the execution is continuously recorded into a log. This log can later be used to reexecute the application, locally or on a different machine, and optionally deploy additional checks during replay, for example for security auditing purposes. An example in this category is Paranoid Android, a record-and-replay system which can efficiently deploy even heavyweight security analyses when replaying mobile apps' execution in the cloud [62].

The idea of combining ASLR [5] with MVX, allowing for non-overlapping layouts to combat code-reuse (and other absolute address-based) attacks, was first proposed by Cox et al. [23]. *MvArmor* extends these techniques to build a new MVX-aware variant strategy which allows complementary per-variant allocators to control memory object allocation in a fine-grained way and effectively counter arbitrary memory error exploits that rely on both absolute and relative object locations. *MvArmor* could be complemented with other memory layout modification strategies, such as fine-grained randomization. Fine-grained randomization [39], [48] has been previously proposed as a comprehensive defense solution against arbitrary memory error exploits, but has proven to be ineffective on its own against modern information disclosure attacks that can bypass any form of ASLR altogether [46].

Finally, the use of hardware-assisted virtualization to sandbox individual processes (granting them access to privileged CPU features) was first proposed by Dune [28], which also forms the basis of *MvArmor*.Hardware-assisted virtualization has also been used to isolate parts of the operating system itself [63] and to facilitate libOS implementations [64], [29], [30]. *MvArmor* draws from prior research in both directions, on one hand, relying on virtualization to efficiently and securely isolate the MVX monitor from untrusted execution, and on the other hand, exploiting libOS-style optimizations to further mitigate the performance impact of traditional MVX implementations.

IX. Conclusion

In this paper, we presented a new design for secure yet efficient MVX systems. Our MVX monitor design leverages hardware-assisted process virtualization to securely and efficiently gain full control over the running program variants. We complemented our design with a new MVX-aware variant generation strategy, which improves the performance and security guarantees of all the prior MVX proposals, resulting in a much more efficient and comprehensive defense solution. Our end-to-end design effectively combines the comprehensive protection against arbitrary memory error exploits provided by fine-grained ASLR strategies with the strong attack detection and disclosure-resistant guarantees provided by MVX.

We implemented our ideas in *MvArmor*, a new secure and efficient MVX system. *MvArmor* demonstrates that many of the performance and/or security limitations of existing MVX solutions are not fundamental and can be effectively addressed with a careful design. *MvArmor*'s policy-driven detection strategy can provide strong and flexible security guarantees at the cost of relatively low run-time overhead for such a comprehensive security solution. Even more surprisingly, *MvArmor* can match the performance of the fastest MVX implementation available while providing far stronger security. Finally, based on a design particularly amenable to optimizations, we believe our framework can provide new opportunities to further enhance the performance of MVX systems. To foster further research in the area and in support of open science, we are making our *MvArmor* prototype available as open source, available at `http://github.com/vusec/mvarmor`.

X. Acknowledgements

We would like to thank the anonymous reviewers for their comments. This work was supported by the European Commission through project H2020 ICT-32-2014 "SHARCS" under Grant Agreement No. 644571 and by the Netherlands Organisation for Scientific Research through the NWO 639.023.309 VICI "Dowsing" project and the NWO "Re-Cover" project.

References

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *S&P*, 2013.

[2] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos, "Memory errors: the past, the present, and the future," in *RAID*, 2012.

[3] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson, "The matter of heartbleed," in *IMC*, 2014.

[4] Microsoft, "A detailed description of the Data Execution Prevention (DEP) feature," http://support.microsoft.com/kb/875352, 2006.

[5] PAX Team, "PAX Address Space Layout Randomization," https://pax.grsecurity.net/docs/aslr.txt.

[6] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattle, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks," in *USENIX SEC*, 1998.

[7] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *S&P*, 2014.

[8] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *CCS*, 2014.

[9] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for C," in *PLDI*, 2009.

[10] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "CETS: Compiler enforced temporal safety for C," in *ISMM*, 2010.

[11] D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for C with very low overhead," in *ICSE*, 2006.

[12] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *USENIX SEC*, 2009.

[13] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity: Principles, implementations, and applications," *ACM TISSEC*, 2009.

[14] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *USENIX SEC*, 2013.

[15] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *USENIX SEC*, 2014.

[16] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained control-flow integrity through binary hardening," in *DIMVA*, 2015.

[17] V. van der Veen, E. Göktaş, M. Contag, A. Pawloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level practical context-sensitive CFI," in *S&P*, 2016.

[18] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *USENIX SEC*, 2015.

[19] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: overcoming control-flow integrity," in *S&P*, 2014.

[20] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *USENIX SEC*, 2014.

[21] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *USENIX SEC*, 2014.

[22] P. Hosek and C. Cadar, "VARAN the unbelievable: An efficient N-version execution framework," in *ASPLOS*, 2015.

[23] 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 SEC*, 2006.

[24] B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space," in *EuroSys*, 2009.

[25] D. Bruschi, L. Cavallaro, and A. Lanzi, "Diversified process replicae for defeating memory error exploits," in *IPCCC*, 2007.

[26] S. Volckaert, B. De Sutter, T. De Baets, and K. De Bosschere, "GHUMVEE: efficient, effective, and flexible replication," in *FPS*, 2012.

[27] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic memory safety for unsafe languages," in *PLDI*, 2006.

[28] A. Belay, A. Bittau, A. J. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged CPU features," in *OSDI*, 2012.

[29] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A protected dataplane operating system for high throughput and low latency," in *OSDI*, 2014.

[30] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," in *OSDI*, 2014.

[31] P. Hosek and C. Cadar, "Safe software updates via multi-version execution," in *ICSE*, 2013.

[32] M. Maurer and D. Brumley, "Tachyon: Tandem execution for efficient live patch testing." in *USENIX SEC*, 2012.

[33] N. Provos, "Improving host security with system call policies," in *USENIX SEC*, 2003.

[34] B. Buck and J. K. Hollingsworth, "An API for runtime code patching," *International Journal of High Performance Computing Applications*, 2000.

[35] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent dynamic instrumentation," in *VEE*, 2012.

[36] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *SOSP*, 1993.

[37] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *CCS*, 2007.

[38] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *USENIX SEC*, 2005.

[39] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Efficient techniques for comprehensive protection from memory error exploits." in *USENIX SEC*, 2005.

[40] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *USENIX SEC*, 2015.

[41] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming," in *S&P*, 2015.

[42] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *USENIX SEC*, 2013.

[43] V. van der Veen, D. Andriesse, E. Goktas, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive CFI," in *CCS*, 2015.

[44] L. Soares and M. Stumm, "FlexSC: Flexible system call scheduling with exception-less system calls," in *OSDI*, 2010.

[45] `https://wiki.ubuntu.com/Security/Features`.

[46] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *S&P*, 2013.

[47] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *OSDI*, 2014.

[48] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization." in *USENIX SEC*, 2012.

[49] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries," in *NDSS*, 2015.

[50] D. Dhurjati and V. Adve, "Efficiently detecting all dangling pointer uses in production servers," in *DSN*, 2006.

[51] P. Akritidis, "Cling: A memory allocator to mitigate dangling pointers." in *USENIX SEC*, 2010.

[52] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: efficient deterministic multithreading in software," *ASPLOS*, 2009.

[53] A. Aviram, S.-C. Weng, S. Hu, and B. Ford, "Efficient system-enforced deterministic parallelism," in *OSDI*, 2010.

[54] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, "DMP: deterministic shared memory multiprocessing," in *ASPLOS*, 2009.

[55] J. Bonwick, "The slab allocator: An object-caching kernel memory allocator," in *USENIX Summer*, 1994.

[56] J. Bonwick and J. Adams, "Magazines and Vmem: Extending the slab allocator to many CPUs and arbitrary resources," in *USENIX ATC*, 2001.

[57] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation – a Pin-based memory characterization of the SPEC CPU2000 and SPEC CPU2006 benchmark suites," *VSSAD Technical Report*, 2007.

[58] A. Avižienis and L. Chen, "On the implementation of N-version programming for software fault tolerance during execution," in *COMPSAC*, 1977.

[59] M. K. Joseph and A. Avižienis, "A fault tolerance approach to computer viruses," in *S&P*, 1988.

[60] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley, "A concurrent dynamic analysis framework for multicore hardware," in *OOPSLA*, 2009.

[61] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart, "Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads," *Technical paper. LMAX*, 2011.

[62] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid android: versatile protection for smartphones," in *ACSAC*, 2010.

[63] R. Nikolaev and G. Back, "VirtuOS: an operating system with kernel virtualization," in *SOSP*, 2013.

[64] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. R. Lorch, B. Bond, R. Olinsky, and G. C. Hunt, "Composing OS extensions safely and efficiently with Bascule," in *EuroSys*, 2013.