I'll Be There for You! Perpetual Availability in the A^8 MVX System

André Rösti
University of California, Irvine
Irvine, CA, USA
aroesti@uci.edu

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

Michael Franz
University of California, Irvine
Irvine, CA, USA
franz@uci.edu

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

***Abstract*—Multi-variant execution (MVX) is a low-friction approach to increase the security of critical software applications. MVX systems execute multiple diversified implementations of the same software in lockstep on the same inputs, while monitoring each variant's behavior. MVX systems can detect attacks quickly and with high probability, because low-level vulnerabilities are unlikely to manifest in precisely the same manner across sufficiently diversified variants. Existing MVX systems terminate execution when they detect a divergence in behavior between variants.**

In this paper, we present A^8, which we believe is the first full-scale survivable MVX system that not only detects attacks as they happen, but is also able to recover from them. Our implementation is comprised of two parts, an MVX portion that leverages the natural heterogeneity of variants running on diverse platforms (*ARM64* and *x86_64*), and a checkpoint/restore portion that periodically creates snapshots of the variants' states and forces variants to roll back to those snapshots upon detection of any irregular behavior. In this way, A^8 achieves availability even in the face of continuous remote attacks.

We consider several design choices and evaluate their security and performance trade-offs using microbenchmarks. Chiefly among these, we devise a system call interposition and monitor implementation approach that provides secure isolation of the MVX monitor, minimal kernel changes (small privileged TCB), and low overheads – a combination not before seen in the context of MVX. We also perform a real-world evaluation of our system on two popular web servers, *lighttpd* and *nginx*, and the database server *redis*, which are able to maintain 53%-71% of their throughput compared to native execution.

1. Introduction

Even after decades of research, memory safety vulnerabilities continue to be a serious threat to software security [1]. Memory-unsafe languages such as C and C++ are *still* among the top 10 most popular languages despite the fact that memory-safe alternatives exist (e.g., Rust) [2]. Consequently, both legacy and new software written in memory-unsafe languages is prone to memory errors [3]. Attackers can exploit memory errors to seize control of even the most *battle-tested* software [4]–[12].

Software diversity, one of the established mitigations against memory exploits, randomizes implementation aspects of programs to generate multiple semantically equivalent program variants [13]–[15]. This forces attackers to target a particular variant, since exploits that successfully compromise one variant often fail at compromising other variants. However, software diversity provides only probabilistic protection, and is bypassable in the presence of information leakages [16].

Multi-variant eXecution (MVX) systems amplify the effectiveness of software diversity techniques by running *multiple* diversified variants in lockstep, typically synchronizing at the granularity of system calls, while feeding them the same inputs [17]–[32]. Previous work has shown that well-constructed variants exhibit the same system call behavior under normal operating conditions, and diverge *only* under attacks [19], [23]. Upon detecting a divergence, the MVX system assumes the variants are under attack. Traditional MVX systems then halt all variants immediately to prevent damage to the host system and its data. Recent research has suggested the use of heterogeneous hardware as a strong source of diversity for MVX systems [33]–[35].

MVX combines a set of unique attributes:

1) *General-purpose:* Unlike specialized mitigations for narrow classes of specific vulnerabilites, MVX systems can detect broad classes of attacks, even some zero-day attacks, as long as attacks manifest themselves differently across variants. MVX also provides a clear-cut approach to progressively raising security: Generating more variants, or increasing the inter-variant variability, leads to immediate gains.
2) *Non-invasive:* Creating variants for MVX in general does not require specialized toolchain support or manual programmer intervention. Alternative ap-

This research was funded in part by the Defense Advanced Research Projects Agency under grants 140D04-23-C-0063 and 140D04-23-C-0070, and the Office of Naval Research under grant N00014-21-1-2409; the opinions expressed here are those of the authors and do not necessarily reflect those of the funding agencies.

proaches, such as software fault isolation (SFI), may require using different tooling, binary rewriting [36], prohibitive amounts of manual effort, or they may require rewriting code to use only a smaller set of allowable features (e.g., no self-modifying code [37]).

3) *Utilizes idle hardware:* Creating efficient multi-threaded software is challenging. As a result, multi-core hardware often remains underutilized. MVX systems are a drop-in solution that can turn these idle resources into increased system security.

MVX systems are comprehensive defenses that intend to protect *critical* infrastructure such as web servers. Such long-running programs cater to an audience with expectations of availability. Yet, all existing MVX systems immediately halt the execution of the variants upon detection of divergences.

In this paper, we present A^8 [1], the first MVX system that detects memory exploits without sacrificing availability – that is, execution of the target program may continue even when a divergence occurs. We opt for a distributed MVX design to improve security by harnessing platform heterogeneity of the participating hosts [33]–[35]. Upon detected divergences, we use checkpoint/restore techniques to restore the running variants to a previous *safe* state instead of terminating their execution. We also highlight overhead and safety considerations of our system by investigating various design and implementation strategies.

In summary, our contributions are the following:

- The design and implementation of A^8, the first MVX system that utilizes platform heterogeneity for security and checkpoint/restore facilities for availability.
- An exploration of different novel strategies for system call interposition, cross-checking, I/O replication, and checkpointing, and a detailed discussion of their trade-offs between security and performance.
- A comprehensive evaluation of our system using microbenchmarks and three popular server applications.

A^8 is available at *https://github.com/andrej/a8.*

2. Background

Our work builds on three main pillars of previous research: We use *software diversity* principles to generate program variants that behave differently under attack. *Multi-variant execution* then allows us to detect these differences at runtime. Upon detected divergences, *checkpoint/restore* facilities assure survivability.

2.1. Software Diversity

Introducing *diversity* can complicate attacks by creating an unpredictable target. One approach to increase diversity is N-version programming [38], [39]. In N-version programming, multiple teams independently construct program implementations based on the same software specification.

Less labor-intensively, compilers, linkers, and operating systems can *automatically* generate program variations at various stages during the deployment cycle [15]. For example, the location of functions can easily be randomized at compile time.

Hardware heterogeneity provides another strong source of diversity: Differences in Instruction Set Architectures (ISAs) and Application Binary Interfaces (ABIs) lead to diverse program layouts, and complicate any attacks relying on details like endianness, register set, struct layout, and available system calls. By compiling software for diverse platforms, we can thwart attacks such as return-oriented programming [40], [41].

2.2. Multi-Variant Execution

MVX systems execute diversified but semantically equivalent program *variants* on identical inputs, and periodically cross-check program behavior at so-called *rendezvous points* using a *monitor* component. Previous work showed that monitoring system calls suffices for security purposes [19], [23]. Consequently, most MVX systems use system calls as rendezvous points, though other options are possible [42], [43].

To ensure security, the program variants must be generated such that they behave identically when provided with benign inputs, but diverge when processing malicious input. An effective way to defend against code-reuse attacks, for example, is to run two variants with *asymmetric address space layouts* [19], [28]: any virtual memory address that points to executable code in one variant refers to data or an unmapped page in the other variant. Consequently, any exploit that relies on absolute code addresses causes divergent behavior, since no address points to code in both variants.

2.2.1. Monitor Component. The monitor component intercepts and cross-checks system calls, and replicates inputs and outputs between variants. It can be implemented as a separate process (so-called *cross-process* monitor), inside the kernel, inside the to-be-monitored process (so-called *in-process* monitor), or as a hybrid combination. *Cross-process* monitors offer strong security guarantees thanks to the operating system's process isolation. However, they introduce significant performance overhead due to the additional required context switches between the monitor and the monitored process, and TLB flushes [44], [45]. *In-kernel* monitors provide the lowest overhead but increase the size of the trusted computing base (TCB). Only a small portion of an MVX monitor requires elevated privileges to operate; including all monitor code in the kernel violates the principle of least privilege and gives any potential vulnerabilities contained in the monitor unnecessarily high potential impact. Another option are *in-process* monitors: While these are efficient, it is challenging to secure and isolate them from the

1. Short for *AAAAAAAA*: Apparatus Assuring Applications Are Always Available Amid Attacks

potentially exploitable to-be-monitored program. As such, their application has been mostly limited to MVX systems focused on *reliability*, i.e., protecting against inadvertent faults rather than deliberate attacks.

2.2.2. I/O Replication. Multi-variant execution must ensure that the executing variants are fed the same inputs, and that they look and behave externally like a single program instance. To do so, the MVX monitor replicates outward-facing resources which cannot be duplicated for each variant. Concretely, under MVX, we must execute certain system calls only once, instead of in each variant, and replicate the results. We refer to these system calls as *non-repeatable system calls*.

Socket-related system calls such as `accept` are one example. In a server application, once a program accepts a client's connection, any subsequent `accept` calls return a different connection. This means that we cannot execute the `accept` call in each program variant. To handle non-repeatable system calls, most MVX systems implement a leader/follower design. A designated leader node is the only one permitted to execute non-repeatable system calls. Follower variants use the system call results from the leader variant.

2.2.3. Distributed MVX. Traditional MVX systems run all variants in parallel on the same physical machine. Distributed MVX systems, on the other hand, execute variants on multiple physical hosts connected through a low-latency network [33]–[35]. These systems exploit ABI/ISA-heterogeneity in the hosts to achieve finer-grained diversity, and hence better security. Distributed MVX systems can protect programs against exploits that traditional centralized MVX systems cannot, e.g., position-independent code reuse [46] and data-only attacks [8]. This justifies the higher overhead associated with a distributed setting [33]–[35].

2.3. Checkpoint and Restore

Checkpoint/restore is a widely used mechanism in distributed systems as a means to enable fault-tolerance [47]–[50]. As a program executes, we can store snapshots of its current state (so-called *checkpoints*). Upon failure, we can resume execution from the previously saved state. This masks the failure to other components and the user. Typically, checkpointing is expensive, since we need to write the memory image of a process to disk. We can use techniques such as incremental and copy-on-write checkpointing to reduce such overheads [51].

A checkpointing mechanism must capture all program state to continue execution upon restoration. This state includes registers, memory contents, and other resources. For a fully consistent picture, process-external entities such as contents of opened files and child processes must also be "dumped". *CRIU* (Checkpoint/restore in Userspace) is a widely used tool that facilitates this in Linux [52]. *CRIU* refuses to create a checkpoint if it cannot restore any of the resources utilized by the process, e.g., opened external TCP connections, in a consistent manner. For many purposes, such a complete checkpoint is not required for a successful restart. For example, in a web server, it may be acceptable to lose some of the opened TCP connections upon restore, since modern web servers incorporate provisions to handle unexpectedly dropped connections. This allows for lighter-weight partial checkpoints.

3. Threat Model

Our threat model is in line with recent work [33]–[35], [44]. We assume an adversary that interacts with the protected target program via a remote connection, such as a network socket. The adversary's goal is to exploit one or multiple memory vulnerabilities present in the target program. A^8 consists of a number of interconnected hosts executing program variants along a monitor. The communication between these hosts takes place on a private network as shown in Figure 1, and is inaccessible to the attacker – the adversary can only interact with the system through a network socket with one designated leader host.

On each of the hosts, the operating system and the hardware are considered part of the trusted computing base (TCB) and are not protected by our system. Consequently, attacks that target the underlying hardware are out of scope for this paper [53]–[57]. We assume that the program variants are sufficiently diverse, and any attempted attack that exploits vulnerabilities of the target program causes the running variants to diverge during their execution. There are several means of generating diverse program variants, and we focus on diversity induced by using heterogeneous hosts.

4. General Design

The goal of A^8 is to execute a long-running program safely, despite the presence of memory vulnerabilities. Whenever an adversary launches an exploit, A^8 detects the resulting divergences and restores the program to a previous safe state. We achieve this through a distributed MVX design coupled with a checkpoint/restore mechanism. In this section, we outline the components of our system and their interactions.

A^8 consists of several heterogeneous, interconnected hosts, as shown in Figure 1. Initially, we derive several variants of the to-be-protected program by compiling it for the ABI/ISA of each host. We then execute these variants concurrently and in lockstep. Each host runs a monitor component alongside its program variant. At regular intervals (defined by the user of our system), the monitors create checkpoints of the variant's current state.

Whenever a variant attempts to execute a system call, its monitor communicates with the monitors of the other variants to ensure that all hosts attempt to execute the same system call. If the monitors do not detect any divergences in the system call numbers or arguments during this cross-checking phase, they allow the system call, replicate the call

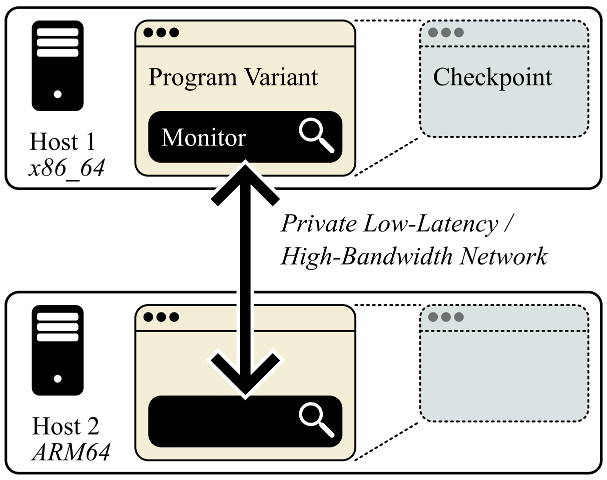

Figure 1: A^8's main components. Two or more diverse hosts execute program variants, while a monitor cross-checks for divergences. The monitor also periodically creates snapshots of the current program state to reset to upon a divergence.

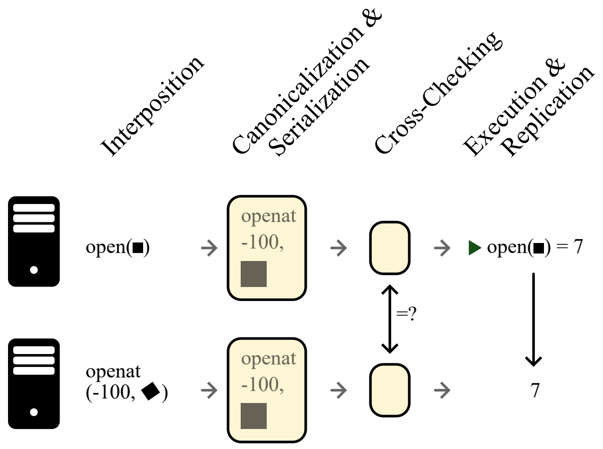

Figure 2: We process system calls in several stages. First, the monitor intercepts the system call. It then serializes the system call number and arguments, and transforms this information into a platform-agnostic form. Subsequently, the monitors communicate to compare these canonical states. If the states match, the monitors resume all variants and replicate the system call results to other variants where needed.

results to other hosts if necessary, and resume execution of the variant. The monitors treat any divergence as an indicator of an ongoing attack or a failure. They address the divergence by resetting the variants to a safe previous checkpoint. This allows the program to continue serving new users, even after attacks or failures.

In our design, we distinguish one designated leader host and one or more follower hosts. Most system calls are executed on all nodes, but the leader is the only one to execute system calls that cannot be repeated (see Section 2.2), replicating their results to followers.

Our MVX system runs variants on heterogeneous hosts that have different ISAs and ABIs. By compiling programs for different platforms, we automatically achieve a high degree of diversity due to differences in low-level details like function sizes and data structure layouts. Our current implementation is geared towards *x86_64* and *ARM64* hosts.

5. Design: Multi-Variant Execution System

Our MVX system fulfills its duties of cross-checking variant behavior and input/output replication primarily by monitoring and manipulating system calls, as shown in Figure 2.

5.1. Secure System Call Interposition

To intercept and manipulate system calls, we implemented a new *safe* in-process monitor, comprised of two cooperating components: a loadable kernel module and a shared library. We embed the MVX monitor into each variant process by forcing it to load our shared library. Once loaded, our kernel module intercepts all system calls and immediately transfers control to a trampoline function within our shared library. In this trampoline function, the in-process monitor can inspect and manipulate the original system call as it sees fit. Once the trampoline function returns, our kernel module ensures that the shared library portion of our monitor is protected from tampering by the rest of the program, and resumes the target program's execution. We kept the size of the kernel module as small as possible, reducing the amount of trusted, highly privileged code. The kernel module's only responsibilities are intercepting system calls and providing isolation for the in-process monitor, while the heavy-lifting of system call monitoring is done in userspace.

As a result of this design, our system call interposition mechanism avoids the high overhead of cross-process monitors, but still enjoys two of their main benefits: (i) low-privilege execution in userspace, and (ii) proper isolation of the monitor from the rest of the program. While approaches based on binary rewriting [21] can achieve even lower overheads, they cannot provide monitor isolation, and thus are not applicable to security-focused applications where the monitor must remain tamper-proof.

5.1.1. General Principle. To load the userspace portion of our in-process monitor into each variant process, we use `LD_PRELOAD` [58]. During its initialization, the shared library uses a custom system call to inform the kernel module about its own address range and the address of a system call trampoline. The kernel module distinguishes *monitor* and *application* system calls. A monitor system call is any system call that originates from a `syscall` instruction in the shared library's address range. Our kernel module allows monitor system calls to execute without monitoring. For application system calls, our kernel module redirects the instruction pointer to the trampoline and immediately returns to the monitor component in userspace. The userspace portion of our monitor then inspects the system call, and may, if it deems appropriate, execute the system call on behalf of the target program. After processing the original system call, our monitor returns to the target program's original system call site via the kernel module, which restores register and stack state as if the originally requested system call returned. Our design is similar to *Syscall User Dispatch (SUD)*, a feature present in recent Linux kernels [59]. However, unlike SUD our kernel module also protects the in-process trampoline against tampering and circumvention, as we describe in the next section [2].

5.1.2. In-Process Monitor Safety. Since we embed the monitor directly into the variant processes, we take extra precautions to protect the monitor from the potentially compromised variant program. A naive implementation allows a malicious target program to circumvent any protections provided by the monitor, either by modifying the monitor's code and data, or by abusing the monitor as a confused deputy to execute unmonitored system calls by jumping directly to a `syscall` instruction in the monitor. We implemented two versions of a safety mechanism that protects our in-process monitor, detailed in the next two sections.

5.1.3. `mprotect`-Based Mechanism. The first mechanism uses hardware memory protection features. We disable the read, write, and execute permissions for all of the monitor's shared library pages. When a system call occurs, our kernel module temporarily restores the original page permissions, just before transferring control to the system call trampoline. When the trampoline returns control to the module, we make the monitor pages inaccessible again. This mechanism works well but incurs high runtime overhead due to additional mode switches and TLB flushes.

5.1.4. Flag/Compare-Based Mechanism. Our second version, shown in Figure 3, vastly improves performance. It protects the integrity of our monitor using three measures:

First, our kernel module rejects any system calls (such as `mprotect`) that attempt to change the protection flags of the monitor's executable or read-only memory. This ensures the integrity of monitor code and read-only data.

Second, whenever the monitor returns from its system call trampoline, our kernel module copies the shared library's mutable memory into a kernel-space shadow buffer. Before the kernel module calls the trampoline, it compares the shadow buffer with the monitor's writable memory. If it detects any changes, the variant tampered with the monitor state, and handles this like any other attack detected by our system through the checkpointing mechanism. This protects the mutable memory of the monitor between system call invocations.

2. https://docs.kernel.org/admin-guide/syscall-user-dispatch.html#security-notes

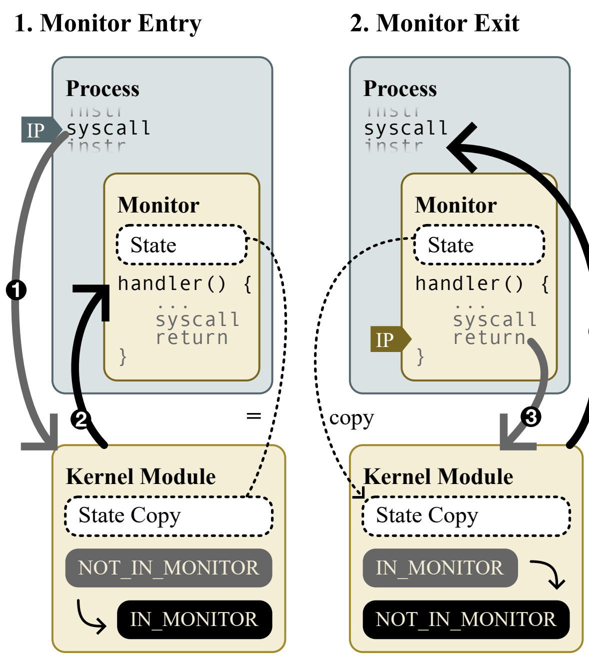

Figure 3: When a target program attempts a system call, the kernel module flips a flag and ensures the monitor's mutable memory has not changed since the last system call invocation by comparing it with a shadow copy in the kernel. The kernel module then forwards the system call information to the in-process monitor. During monitor execution, system calls are unmonitored, allowing the monitor to execute the requested system call on behalf of the program. Upon monitor return, the kernel module flips the flag back and updates the shadow copy of monitor state before returning to the original system call site.

Third, we protect against unauthorized execution of system call instructions in the monitor's code using state tracking. The kernel module sets an `IN_MONITOR` flag whenever it redirects execution to the monitor trampoline. We reset this flag when the trampoline returns. If the target program executes a `syscall` instruction from one of the monitor's code pages, and the flag is not set, we assume the variant is trying to abuse our monitor as a confused deputy, and immediately restore to the last checkpoint. Thus, the userspace portion of the monitor can only execute system calls if execution properly flowed to its trampoline via the kernel.

5.2. Canonicalization & Serialization

After system call interception, our userspace monitors communicate with each other to verify that all variants called equivalent system calls. To do so, the monitors first serialize and canonicalize the system call information into a platform-agnostic normalized form.

5.2.1. Canonicalization. OS resource identifiers, such as file descriptors (FDs) and process IDs (PIDs), may differ between variants even if they refer to the same underlying resource. Our system maintains a second set of *canonical* identifiers, and a mapping between canonical and local identifiers. The target program is only ever exposed to canonical identifiers; we perform translation to and from local identifiers at system call entry and exit.

Besides identifier canonicalization, system call numbers and arguments also need to be translated into a platform-agnostic form. First, we redirect system calls that do not exist in all architectures. For example, in Linux for the *ARM64* architecture, many legacy system calls are missing. These must be redirected in *x86_64* variants. When a legacy system call is observed, we use its newer superseding system call as the canonical form. Second, system call argument data types may differ in alignment and size between architectures. We canonicalize such data types to a platform-independent form with fixed data type sizes and consistent padding/alignment. For example, we implemented such canonicalization for the `struct stat` and `struct epoll_event` data structures.

5.2.2. Serialization. For transmission, we must serialize the system call arguments, including any referenced buffers. Where replication is needed, we also serialize the results of system calls. We implemented a generic framework, in which the system call entry handler provides a *type description* for each of its arguments (or return values). Types can be `IMMEDIATE`, `POINTER`, or `BUFFER`.

`POINTER` type descriptions specify the type of the pointed-to data, often a `BUFFER`. `BUFFER` type descriptions indicate the length of the buffer, and any number of references contained within the buffer. References are tuples of (`offset,type`), which describe pointers within the buffer that need to be recursively serialized.

The system call entry handlers dynamically generate these type descriptions, because not enough information is available statically to describe all argument types. For example, for many system calls such as `write`, one argument is a buffer, whose size is given dynamically in one of the other arguments.

5.3. Cross-Checking

Our cross-checking mechanism verifies that all variants execute semantically equivalent system calls.

5.3.1. Cross-Checking Policy. Since cross-checks require (costly) network communication between all hosts, omitting cross-checking for non-critical calls can greatly reduce overhead. We thus allow the system administrator to configure a cross-checking policy, which exempts some system calls from cross-checking. Different policies provide different trade-offs between security and performance. The available policies are identical to previous work [34], [44]. We list a selection of three interesting policies with their checked system calls and the class of attacks they protect from, in increasing order of security:

- *Code Execution*: `execve`, `mmap`, `mprotect`
- *Information Disclosure*: read/write calls, `execve`, `mmap`, `mprotect`
- *Comprehensive* - cross-checks all system calls

5.4. Execution & Results Replication

After successful cross-checking, we either execute the system call on all nodes, on the leader node only, or we emulate the system call. We execute system calls inside the monitor, on behalf of the traced program variant. This "delegation" approach [44], [60] prevents time-of-check-to-time-of-use attacks, and it allows emulation, which we use to quickly return cached results for system calls such as `getpid`.

We execute non-repeatable system calls (see Section 2.2) on the leader only and replicate the system call return value to all follower nodes. We do this using the same canonicalization & serialization framework described before. To the program, the result looks as if the system call executed locally.

Replication is one of the biggest sources of overhead in our system, because it requires network transmissions. We thus avoid results replication and execute system calls locally on each node when possible.

5.4.1. Replication Batching. In contrast to cross-checking, which can be omitted for non-security-sensitive system calls, replication can never be skipped; the target program depends on system call results to continue execution.

However, to improve replication performance, we can batch replication information. When batching is enabled, the leader gathers the results of back-to-back non-cross-checked system calls locally. When we reach threshold batch size (or a system call requires cross-checking), the leader sends its batch to followers. Thus, the leader progresses ahead of the follower in program execution for a number of system calls. When the leader sends replication information, the followers catch up.

Although batching increases the latency of some system calls individually, it improves the overall system performance, because it amortizes network communication overheads. To see this, consider a chain of N back-to-back non-cross-checked system calls, each requiring results replication. Each system call i requires processing time on the leader p_i^{leader} and the followers p_i^{follower}. Every network exchange j introduces a fixed latency overhead c^{latency} and a length-dependent transmisison time $c_j^{\text{transmission}}$.

Without batching, the time required for our chain of N system calls on the follower is:

$$Nc^{\text{latency}} + \sum_{i=1}^{N} p_i^{\text{leader}} + c_i^{\text{transmission}} + p_i^{\text{follower}}$$

Note that we have N individual network exchanges. Now, if we batch up N system call results into one network exchange, execution time becomes:

$$\underbrace{\left(\sum_{i=1}^{N} p_i^{\text{leader}}\right) + c^{\text{latency}} + c_{1..N}^{\text{transmission}}}_{\text{wait for leader to execute \& replicate syscalls 1 .. N}} + \left(\sum_{i=1}^{N} p^{\text{follower}}\right)$$

Note that now we have only *one* (larger) network exchange, so we save $(N-1)$ terms of c^{latency}. Since many to-be-replicated system call return values are only a few bytes in size (often a single integer indicating error or success), exchanging such small values individually quickly leads to network latency dominating execution overheads, making batching beneficial.

5.4.2. Other Optimizations. Asynchronous replication exploits the fact that we can exchange replication information in a "fire-and-forget" manner. With asynchronous replication enabled, the leader does not wait for the followers' receipt acknowledgements. Furthermore, we identified cases where we can cache replication results. We found the contents of many replication buffers to be identical, e.g., when a program repeatedly reads the same file. We considered keeping a local cache of frequently encountered replication buffers. If a replication buffer is resident in the cache, the leader sends a small identifier for the cache entry number instead of its contents. However, due to the small size of the individual replication buffers we encountered in our benchmarks, caching was not a useful optimization; the additional processing slowed the system down overall. Caching may, however, be useful when using a slower inter-monitor connection.

6. Design: Checkpoint/Restore

Our checkpoint/restore mechanism is the key enabler for the survivability of the applications we protect. At regular intervals, our system captures the state of all variants. We then roll all variants back to the most recent checkpoint if we detect divergent behavior, as shown in Figure 4.

6.1. Determining Checkpointing Locations

Our system creates checkpoints at configurable program-specific locations. The system administrator must carefully choose locations that are (i) functional and (ii) safe. For functionality, any location that allows the program to continue executing without errors is acceptable. For safety, we should avoid any locations at which an attacker may have already compromised the system. Consider that an attack may begin at some point in time t_a, but not be detected until a later system call at time t_b. Creating a *compromised* checkpoint during the attack build-up (i.e., at time t with $t_a \leq t < t_b$) could result in denial of service, as our system would repeatedly detect the attack, then reset the system only to launch the same attack again. We must thus choose

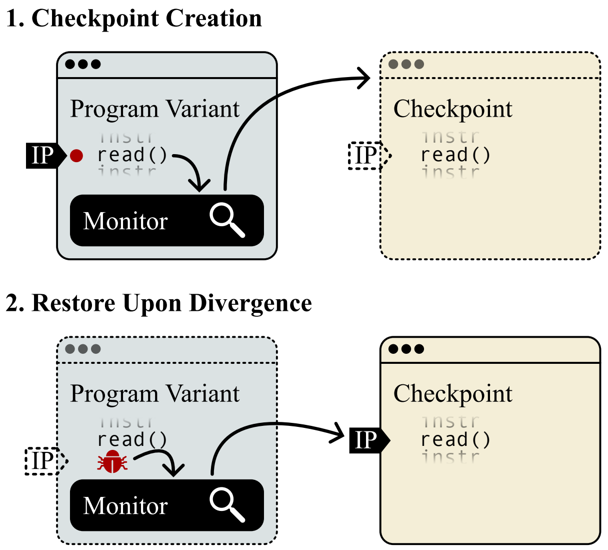

Figure 4: At user-defined breakpoints, the monitor creates a copy of the current program state. When the monitors observe divergent behavior (e.g., due to an implementation bug or attack), they roll execution of all variants back to the most recent checkpoint.

a checkpointing location for which we can reason, based on program-specific semantics and our threat model, that it is a "clean slate".

In general, we have observed that long-running programs often contain some variation of a main loop that polls for events or requests. The beginning of an iteration of such a loop is often suitable as a checkpointing location. In a web server, for example, the connection "accept" phase is a suitable checkpointing location. Rolling back to the beginning of the "accept" phase will result in the server dropping the attacker's connection, and return the server to a state where it is ready to serve the next incoming request. Doing so also ensures safety: Before the remote attacker's connection was accepted, (s)he could not yet have compromised the system.[3]

In our implementation, system administrators define checkpointing locations in a configuraiton file as either a fixed addresses or a symbol name (e.g. a function) in the target executable (plus an optional offset). Upon startup, our monitors insert software breakpoints (`int3` on *x86_64*, `brk #0` on *ARM64*) at those locations. Our monitor also registers a custom `SIGTRAP` signal handler, which the OS invokes whenever a variant hits one of these software breakpoints. The signal handler then initiates our checkpoint creation mechanism, single-steps the original instruction by inserting another software breakpoint directly past it, reinserts the breakpoint at the original instruction, and then resumes normal execution.

3. This argument assumes there are no concurrent connections, which is the case for our benchmarks.

6.2. Checkpoint Creation and Restoration

We implemented two versions of our checkpoint/restore mechanism. The first stores program state to disk and is suited for most programs but is slow. The second only retains the most recent checkpoint in memory and does not support multi-threaded applications but is considerably faster.

6.2.1. *CRIU*-Based Checkpointing. The first version of our checkpoint/restore mechanism is based on *CRIU* [52]. To create a checkpoint, *CRIU* suspends a process and dumps all of its state to an image file on a disk. Creating checkpoints with `CRIU` is relatively slow due to the comprehensive nature of its state capturing and use of disk storage.

6.2.2. `fork`-Based Checkpointing. To address the performance degradation caused by *CRIU*, we implemented an alternative, more lightweight design that trades completeness for speed. This design uses the `fork` system call, `setjmp/longjmp`, and a shared memory block. At a glance, it works in three steps: (i) we create an identical child of the to-be-checkpointed parent process, (ii) we pause execution of that child, and (iii) we resume execution of the child and kill the parent when we want to restore the checkpoint.

Concretely, we implemented this scheme as follows: First, we initialize a shared memory block to be used for communication. When we want to create a checkpoint, we issue a `fork` call in the variant process. The resulting child process is an exact duplicate of its parent, including a copy of the parent's virtual address space, with access to the shared memory. Our checkpointing code in the child then pauses until it receives a command from its parent passed through the preallocated shared memory block.

To restore the last checkpoint, the parent process sends a `restore` command to the child and exits. The checkpointing code in the child, waiting for commands, sees this, and resumes execution where the checkpoint creation routine was first entered using `setjmp/longjmp`. By resuming execution at the beginning of the checkpoint creation routine, the to-be-restored checkpoint is effectively duplicated. One of the copies then continues executing normally, starting from the checkpointed location, while the other is once again paused waiting for commands, allowing us to restore the same checkpoint multiple times. If a checkpoint is no longer needed, the parent can instead also send a `delete` command, which causes the child to exit.

7. Evaluation

We assess the real-world practicality of our system on two popular web servers, *lighttpd* and *nginx*, and a database server, *redis*. We also analyze which components contribute most to our system overheads using four microbenchmarks. Our experiment setup consists of an *ARM64* leader machine (2.0GHz Marvell Thunder X ARMv8 processor, 64 GB DRAM) and an *x86_64* follower machine (3.6 GHz

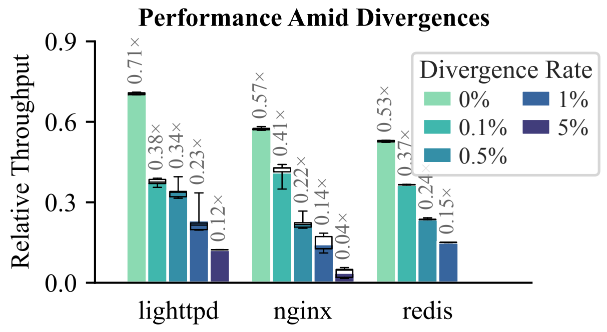

Figure 5: Throughput of our system relative to native execution (higher is better). In the benign setting (0% divergences) our system monitors program execution, but no attacks occur. In the increasingly *hostile* settings ($0.1\% - 5\%$), we randomly inject divergences (simulating attacks or failures). Upon a divergence, our system resets variants to the last created checkpoint. Note that *redis* was unable to serve any requests at the 5% error rate. We used a permissive (*Code Execution*) policy and a batch size of $8KB$ for these experiments.

Intel Core i9, 32 GB DRAM) that are directly connected via Mellanox ConnectX-5 100 Gbps NICs. Both machines run Ubuntu 22.04.4 LTS (leader: kernel version 5.15.151). Dynamic frequency and voltage scaling are disabled, and benchmarks are isolated on their own dedicated cores. We report the mean measurements of five runs; the standard deviation in all cases, except for the experiments shown in Figure 5, is below 2.5%.

7.1. *lighttpd*, *nginx*, and *redis*

To test each of the *lighttpd*, *nginx* and *redis* workloads, we opened 10 client connections and issued back-to-back requests from a separate client machine, which is connected to the leader with a 1 Gbps link. This evaluation setting is similar to previous work [33]–[35]. For the web servers, we used the *wrk* HTTP benchmarking utility; for *redis*, we used *redis-benchmark*, which we slightly modified to ignore failed requests (due to simulated errors). As checkpoint creation locations, we chose the "connection close" phase for *nginx* and *lighttpd*, and the start of the main event loop for *redis*.

Figure 5 shows how our system fares in varyingly hostile real-world settings. Note that A^8 continues serving requests, even after repeated failures; this is not possible in other MVX systems. We injected errors (which trigger divergences) with the given probabilities for each system call. These low per-system-call error rates translate to much higher per-request failure probabilities, since the target programs issue multiple system calls for each handled request. For example, *lighttpd* issues 12 system calls on average per request – for our highest error rate of 5%, this translates to

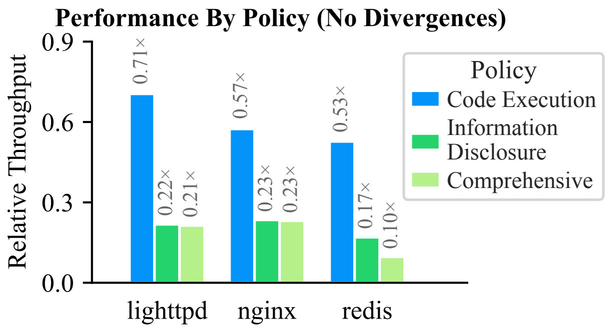

Figure 6: Effect of cross-checking policy on request throughput, relative to native execution (higher is better). The batch size was $8KB$.

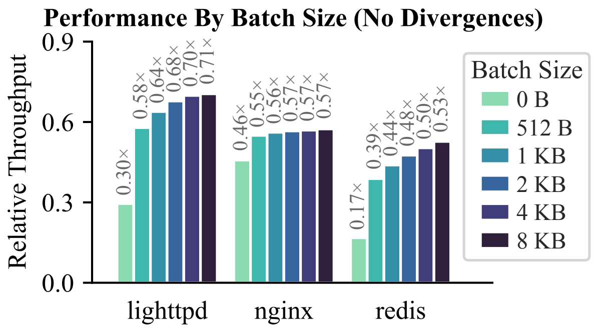

Figure 7: Effect of replication batch size on request throughput relative to native (higher is better). We used the *Code Execution* policy. Lenient policies enable batching replication information (see Section 5.4), which amortizes the cost of network latency.

a probability of $1 - 0.95^{12} = 46\%$ that any given request experiences a failure.

To demonstrate the benefits of deploying our system, we compare it to a simple alternative approach that achieves crash-survivability: *Restarting the target application whenever a failure occurs.*[4] We use *redis* as an example. In our experiments, native *redis* took $0.02ms$ on average to process one request, and its start-up time was $5ms$. *redis* issues three system calls (`epoll_pwait`, `read` and `write`) for each received ping request; if any of those calls fail, the entire request fails. Therefore, if some anomaly causes system calls to fail with a 1% probability, we expect $\sim 3\%$ of all requests to fail.[5] If we crash and restart upon each failure, we expect a new average request processing time of

$$97\% \times 0.02ms + 3\% \times (5ms + 0.02ms) = 0.17ms,$$

i.e., the 3% of requests that fail incur an additional "restart" overhead of $5ms$ each. This equates to roughly $0.11\times$ of native execution throughput without failures. Note that this alternative also does not protect against deliberate attacks, only accidental failures. In contrast, our solution, which includes application monitoring and protection against deliberate attacks, achieves $0.15\times$ of native throughput, at the same 1% per-*syscall* error rate. Our solution also retains the built-up state of the application up to the last created checkpoint, whereas the alternative approach might lose valuable information at each restart. This evidences the advantage of A^8's checkpoint/restore-MVX approach.

If there are no errors during execution, our system performance is comparable to similar MVX systems. For example, *MvArmor* [29] introduces $1.4\times$ and $1.7\times$ runtime overhead for *nginx* and *lighttpd*, respectively, under a *Code Execution* policy with two running variants, whereas A^8 introduces $1.8\times$ and $1.4\times$ runtime overhead for the same benchmarks and setting (for this comparison, we calculated runtime as the inverse of throughput in Figure 6).

Figures 6 and 7 show how the cross-checking policy and replication batch size affect performance overheads when no failures occur (see Sections 5.3 and 5.4). Exempting system calls from cross-checking via a permissive policy greatly reduces overheads in two ways: (i) some costly cross-checking network traffic is eliminated, and (ii) batching of replication information becomes possible. Note that the replication batch size merely is an upper limit; our system will send out replication information before the batch is filled if the next system call requires cross-checking. We observed that larger batch sizes yield no additional benefit under more restrictive policies. The largest effective batch size is directly related to the amount of replication information generated by non-checked system calls between any two checked system calls.

7.2. Microbenchmarks

The primary overheads of our system stem from three sources: (i) system call interposition, (ii) network communication (for cross-checking and result replication), and (iii) checkpoint creation. We evaluated each of those sources in isolation with four simple microbenchmarks, which repeatedly issue one system call (`read` on a device and on a file, `getcwd`, and `sched_yield`) for one million iterations.

7.2.1. System Call Interposition Overhead. We isolated the cost of intercepting and redirecting system calls to our in-process monitor by disabling all other processing (cross-checking, replication, checkpoint creation) in the monitor. The monitor protection mechanism (see Section 5.1.2) then dominates the overhead, as evidenced by Figure 8. The large slowdowns of the `mprotect`-based mechanism on *ARM64* ($14.8 - 20.5\times$) and *x86_64* ($6.9 - 8.1\times$) primarily occur due to the TLB flush incurred when changing page table protection bits. The flag/compare mechanism, which uses an in-kernel shadow buffer to guarantee monitor integrity,

4. A "watchdog", that immediately restarts the program if it crashes, is often present on server applications. This only provides resilience against application crashes. Targeted attacks that divert program execution successfully without a crash remain undetected.

5. Successful request probability: $0.99^3 \approx 97\%$

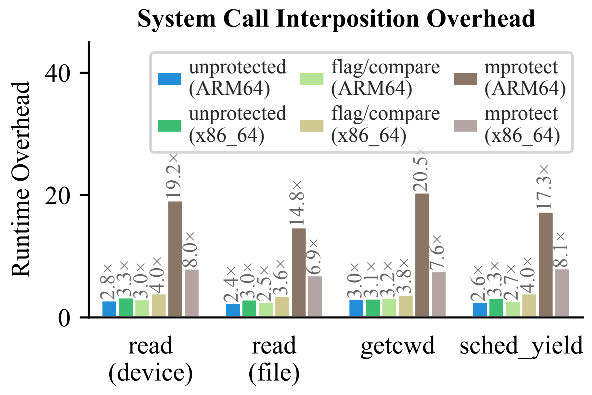

Figure 8: Overhead of interposing and forwarding system calls to our protected in-process monitor by protection mechanism (lower is better).

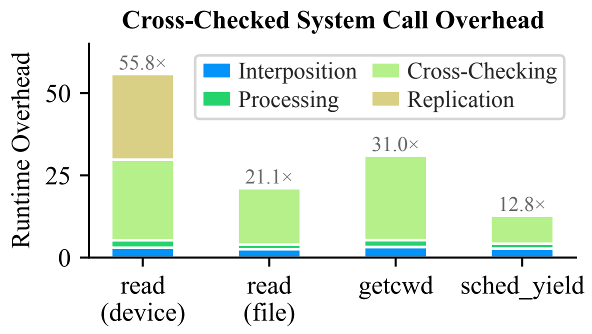

Figure 9: System call overheads with cross-checking across one *x86_64* and one *ARM64* host, relative to native performance of the slower *ARM64* host (lower is better).

incurs negligible runtime performance overheads compared to the unprotected monitor,[6] making this an attractive combination of safety and speed. However, we observed that the cost of this protection approach scales linearly with the size of protected memory. It is, therefore, advisable to minimize the monitor memory usage. We used the flag/compare-based mechanism in all of the following microbenchmarks.

7.2.2. Networking: Multi-Variant Execution Overhead. One of the primary sources of overhead in a distributed MVX system is the network communication required for cross-checking and results replication. We aimed to minimize both the number and size of exchanged messages, and we implemented batching of replication messages to amortize the network latency. Figure 9 shows the total system call execution overhead of the multi-variant execution portion of our system (i.e., checkpointing disabled) for cross-checked

6. An unprotected monitor should not be used for a system like ours, as it can easily be corrupted. We show it here only as a lower-bound.

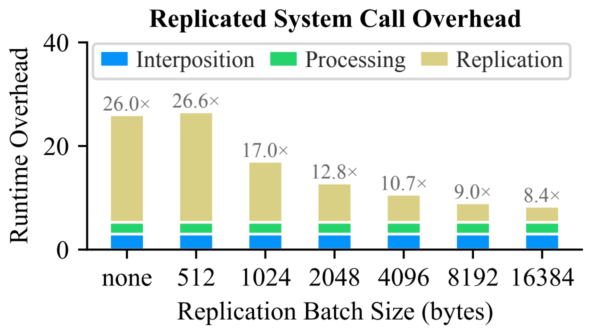

Figure 10: Breakdown of execution overhead for a non-cross-checked and replicated system call, reading 512 bytes from a device (lower is better). Network latency is amortized by executing multiple system calls on the leader, and then replicating those results to the follower in batches.

system calls. All system calls in our system pass through interposition, serialization, and canonicalization phases. Additionally, for this experiment, we activated cross-checking for all system calls. Cross-checking requires exchanging messages between variants, which adds networking overhead. One microbenchmark also required replication; `read` operations on device files are non-repeatable, and therefore only execute on the leader machine, necessitating network communication for result replication. The three other system calls in this experiment executed locally, eliminating the need to replicate results. The cross-checking cost dominates their overhead. In line with prior work [34], we observed that cross-checking between variants incurs high runtime overhead, dominated by network communication costs. Note, however, that these benchmark programs relentlessly issue back-to-back system calls. This represents a worst-case scenario for our system. Real-world applications are unlikely to experience slowdowns as dramatic as this, since their execution also incorporates some userspace processing, which is not slowed down by our system. Further note that system calls can be exempted from cross-checking by selecting a permissive policy (see Section 5.3), in which case overheads consist solely of interposition, processing, and potential replication.

We also evaluated the effect of replication batching with varying batch size (see Section 5.4) using our `read` microbenchmark and present our results in Figure 10. To allow for batching to take effect, we disabled cross-checking, since cross-checking would require flushing the replication buffer. Thus, this microbenchmark captures an ideal case where unlimited back-to-back non-cross-checked system calls take place. In real-world use cases, a system call requiring cross-checking would eventually be encountered, setting an upper limit to the effective batch size. Our evaluation shows that, as batch size increases, we use the internal network more efficiently by exchanging more information per message.

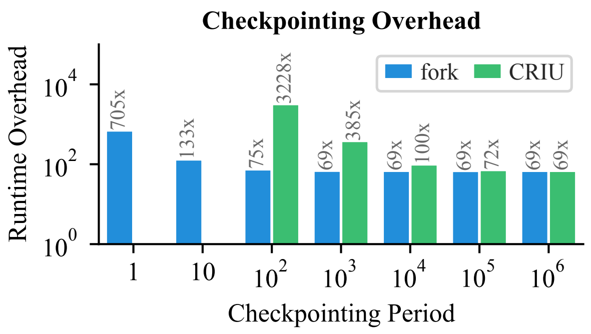

Figure 11: Relative runtime overhead imposed by checkpoint creation at regular intervals in the `read(device)` microbenchmark (log scale, lower is better). At a period of 1, we create a new checkpoint for every `read` issued. At a period of 10^6 we create exactly one checkpoint for the entire program execution, since there are 10^6 total iterations. We did not perform *CRIU* checkpointing for periods of 1 and 10, due its large cost.

7.2.3. Checkpointing Overhead. We evaluated both our `fork`-based and *CRIU*-based checkpointing mechanisms on the `read (device)` microbenchmark. To isolate checkpointing costs, we executed only one variant (thus eliminating network communication costs of MVX). Figure 11 shows the results of this microbenchmark running on the *ARM64* server. We chose the system call site inside the microbenchmark's main loop as the checkpoint creation location. Our results indicate that the `fork`-based checkpointing significantly outperforms *CRIU*-based checkpointing, which we attribute to the following reasons: (i) `fork` uses copy-on-write paging to delay and potentially avoid copying the checkpointed process' memory contents, (ii) this approach stores checkpoint images in memory rather than on disk, (iii) child processes hosting the checkpoint images share much of their process state (e.g., file descriptors) with the checkpointed process, thus allowing us to minimize the amount of state we need to store, and (iv) *CRIU* aims to store all associated state of a process, whereas the fork approach is more limited. Note that the results shown in Figure 11 correspond to an extreme case in which the breakpoint triggering the creation of a checkpoint is inside a hot loop. For more realistic use cases, we should carefully choose breakpoint locations that minimize the number of generated checkpoints, thus reducing the checkpointing cost.

8. Discussion

We envision several avenues for future improvements to A^8: To improve performance, our system call interposition mechanism could use novel hardware-based protection mechanisms such as Intel CET and PKU [61], eliminating the need to check monitor integrity at each system call.

To improve generality, we could extend our system to support multi-threaded applications. Techniques to support multi-threaded programs in MVX already exist [62], [63] and could be ported to our system.

Our implementation currently also requires manual selection of checkpointing locations; the automatic detection of suitable checkpoint locations could also be an interesting avenue of future research.

9. Related Work

9.1. Traditional Mitigations

Both industry and academia proposed two main classes of mitigations against memory exploits, control-flow integrity [36], [64], [65] and software diversity [15]. Operating systems and stock toolchains adopted some of these mechanisms [66], and even CPU manufacturers added hardware support for them (e.g., Intel CET [61]). Unfortunately, both software diversity and control-flow integrity have known weaknesses [8], [46], [67].

9.2. Multi-Variant Execution

Previous work explored multi-variant execution solutions for security [17]–[19], [23], [24], [29], [31], [35], [68] and reliability purposes [21], [25], [69]–[71]. Xu et al. and Pei et al. also showed that MVX systems can be used to combine multiple security mechanisms [63] and reconstruct exploits respectively [72]. Multi-variant execution is not without its flaws, however, as it usually imposes non-trivial requirements on protected programs. Researchers proposed techniques to alleviate limitations that are common to various MVX systems such as thread synchronization [62], [63], shared-memory support [18], [73], address-dependent behavior [27], and consistent signal delivery [18], [23].

9.3. System Call Interposition

Previous research has investigated several techniques for system call interposition [74]–[76]. These techniques include utilizing existing Linux facilities [45], [59], [77]–[80], modifying the kernel [29], [60], [81]–[85], and employing binary rewriting [21], [86]–[88]. Additionally, some studies have explored the integration of various approaches [89], [90].

10. Conclusion

We presented A^8, a system that combines multi-variant execution with checkpoint/restore facilities to protect long-running programs, e.g., server applications, facing continuous attacks without undermining their availability. A^8 harnesses platform heterogeneity to defend against memory exploits. Our system incorporates new mechanisms for safe and fast system call interposition, amortized state replication, and checkpointing. Our evaluation shows promising results on microbenchmarks and popular server applications, demonstrating the potential of our approach to improve the security and availability of critical infrastructure.

References

[1] "CWE," https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html, Accessed 2023.

[2] "Stackoverflow," https://survey.stackoverflow.co/2022/#most-popular-technologies-language, Accessed 2023.

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

[4] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *ACM Conference on Computer and Communications Security (CCS)*, 2007.

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

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

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

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

[9] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *IEEE Symposium on Security and Privacy (S&P)*, 2013.

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

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

[12] N. Yadav, F. Vollmer, A.-R. Sadeghi, G. Smaragdakis, and A. Voulimeneas, "Orbital shield: Rethinking satellite security in the commercial off-the-shelf era," in *IEEE Conference on Security for Space Systems (3S)*, 2024.

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

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

[15] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[16] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, A.-R. Sadeghi, and H. Okhravi, "Address oblivious code reuse: On the effectiveness of leakage resilient diversity," in *Symposium on Network and Distributed System Security (NDSS)*, 2017.

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

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

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

[20] B. Salamat, A. Gal, T. Jackson, K. Manivannan, G. Wagner, and M. Franz, "Multi-variant program execution: Using multi-core systems to defuse buffer-overflow vulnerabilities," in *International Conference on Complex, Intelligent and Software Intensive Systems*, 2008.

[21] P. Hosek and C. Cadar, "Varan the unbelievable: An efficient n-version execution framework," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

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

[23] B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space," in *European Conference on Computer Systems (EuroSys)*, 2009.

[24] S. Österlund, K. Koning, P. Olivier, A. Barbalace, H. Bos, and C. Giuffrida, "kMVX: Detecting kernel information leaks with multi-variant execution," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[25] M. Maurer and D. Brumley, "TACHYON: Tandem execution for efficient live patch testing," in *USENIX Security Symposium*, 2012.

[26] B. Salamat, T. Jackson, C. W. Gregor, Wagner, and M. Franz, "Run-time defense against code injection attacks using replicated execution," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2011.

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

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

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

[30] D. Kim, Y. Kwon, W. N. Sumner, X. Zhang, and D. Xu, "Dual execution for on the fly fine grained execution comparison," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

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

[32] A. Rösti, A. Voulimeneas, and M. Franz, "The astonishing evolution of probabilistic memory safety: From basic heap-data attack detection toward fully survivable multivariant execution," *IEEE Security & Privacy*, 2024.

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

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

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

[36] M. Abadi, M. Budiu, Úlfar. Erlingsson, and J. Ligatti, "Control-flow integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[37] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," *Communications of the ACM*, vol. 53, no. 1, pp. 91–99, 2010.

[38] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, vol. 1, 1978, pp. 3–9.

[39] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering (TSE)*, no. 12, pp. 1491–1501, 1985.

[40] A. Venkat and D. M. Tullsen, "Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 121–132, 2014.

[41] A. Venkat, S. Shamasunder, H. Shacham, and D. M. Tullsen, "Hipstr: Heterogeneous-isa program state relocation," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[42] R. Huang, D. Y. Deng, and G. E. Suh, "Orthrus: Efficient software integrity protection on multi-cores," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

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

[44] S. Volckaert, B. Coppens, A. Voulimeneas, A. Homescu, P. Larsen, B. De Sutter, and M. Franz, "Secure and efficient application monitoring and replication." in *USENIX Annual Technical Conference*, 2016.

[45] M. Gebai and M. R. Dagenais, "Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead," *ACM Computing Surveys (CSUR)*, vol. 51, no. 2, pp. 1–33, 2018.

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

[47] T. Herault and Y. Robert, *Fault-tolerance techniques for high-performance computing*. Springer, 2015.

[48] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on software Engineering*, no. 1, pp. 23–31, 1987.

[49] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala, "Checkpointing and its applications," in *Twenty-fifth International Symposium on fault-tolerant Computing. Digest of papers*. IEEE, 1995, pp. 22–31.

[50] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Transactions on parallel and Distributed Systems*, vol. 9, no. 10, pp. 972–986, 1998.

[51] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "Measured performance of consistent checkpointing," in *Proceedings of the Eleventh Symposium on Reliable Distributed Systems*, no. CONF, 1992.

[52] "Criu," https://criu.org/, Accessed 2023.

[53] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

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

[55] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi, "V0ltpwn: Attacking x86 processor integrity from software," in *USENIX Security Symposium*, 2020.

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

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

[58] "LD_PRELOAD," https://man7.org/linux/man-pages/man8/ld.so.8.html, Accessed 2023.

[59] "Syscall user dispatch," https://docs.kernel.org/admin-guide/syscall-user-dispatch.html, Accessed 2023.

[60] T. Garfinkel, B. Pfaff, M. Rosenblum *et al.*, "Ostia: A delegating architecture for secure system call interposition." in *Symposium on Network and Distributed System Security (NDSS)*, 2004.

[61] Intel, "Intel 64 and ia-32 architectures software developer's manual," 2023.

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

[63] M. Xu, K. Lu, T. Kim, and W. Lee, "Bunshin: compositing security mechanisms through diversification," in *USENIX Annual Technical Conference*, 2017.

[64] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, p. 16, 2017.

[65] N. Burow, X. Zhang, and M. Payer, "Sok: Shining light on shadow stacks," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[66] PaX Team, "Address space layout randomization (aslr)," https://pax.grsecurity.net/docs/aslr.txt, 2001.

[67] grsecurity, "Close, but no cigar: On the effectiveness of intel's cet against code reuse attacks," https://grsecurity.net/effectiveness_of_intel_cet_against_code_reuse_attacks, 2016.

[68] A. M. Espinoza, R. Wood, S. Forrest, and M. Tiwari, "Back to the future: N-versioning of microservices," in *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2022.

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

[70] L. Pina, A. Andronidis, M. Hicks, and C. Cadar, "Mvedsua: Higher availability dynamic software updates via multi-version execution," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[71] L. Pina, D. Grumberg, A. Andronidis, and C. Cadar, "A {DSL} approach to reconcile equivalent divergent program executions," in *USENIX Annual Technical Conference*, 2017, pp. 417–429.

[72] Z. Pei, X. Chen, S. Yang, H. Duan, and C. Zhang, "Taichi: Transform your secret exploits into mine from a victim's perspective," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2023.

[73] J. Vinck, B. Abrath, B. Coppens, A. Voulimeneas, B. D. Sutter, and S. Volckaert, "Sharing is caring: Secure and efficient shared memory support for MVEEs," in *European Conference on Computer Systems (EuroSys)*, 2022.

[74] T. Garfinkel, "Traps and pitfalls: Practical problems in system call interposition based security tools," in *Symposium on Network and Distributed System Security (NDSS)*, 2003.

[75] D. A. Wagner, "Janus: an approach for confinement of untrusted applications," Master's thesis, University of California, Berkeley, 1999.

[76] T. Fraser, L. Badger, and M. Feldman, "Hardening cots software with generic software wrappers," in *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, vol. 2. IEEE, 2000, pp. 323–337.

[77] J. Keniston, A. Mavinakayanahalli, P. Panchamukhi, and V. Prasad, “Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps,” in *Proceedings of the 2007 Linux symposium*, 2007, pp. 215–224.

[78] K. Jain and R. Sekar, “User-level infrastructure for system call interposition: A platform for intrusion detection and confinement.” in *Symposium on Network and Distributed System Security (NDSS)*, 2000.

[79] M. Gülmez, T. Nyman, C. Baumann, and J. T. Mühlberg, “Rewind & discard: Improving software resilience using isolated domains,” in *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2023.

[80] F. Yang, B. Im, W. Huang, K. Kaoudis, A. Vahldiek-Oberwagner, C. che Tsai, and N. Dautenhahn, “Endokernel: A thread safe monitor for lightweight subprocess isolation,” in *USENIX Security Symposium*, 2024.

[81] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard, “Jenny: Securing syscalls for PKU-based memory isolation systems,” in *USENIX Security Symposium*, 2022.

[82] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, P. Druschel, and D. Garg, “Erim: Secure, efficient in-process isolation with memory protection keys,” in *USENIX Security Symposium*, 2019.

[83] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, “Hodor: Intra-process isolation for high-throughput data plane libraries,” in *USENIX Annual Technical Conference*, 2019.

[84] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, “Dune: Safe user-level access to privileged CPU features,” in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[85] A. Voulimeneas, J. Vinck, R. Mechelinck, and S. Volckaert, “You shall not (by)pass! practical, secure, and fast pku-based sandboxing,” in *European Conference on Computer Systems (EuroSys)*, 2022.

[86] P.-A. Arras, A. Andronidis, L. Pina, K. Mituzas, Q. Shu, D. Grumberg, and C. Cadar, “Sabre: load-time selective binary rewriting,” *International Journal on Software Tools for Technology Transfer*, pp. 1–19, 2022.

[87] “Userspace syscall intercepting library,” https://github.com/pmem/syscall_intercept, Accessed 2023.

[88] K. Yasukata, H. Tazaki, P.-L. Aublin, and K. Ishiguro, “zpoline: a system call hook mechanism based on binary rewriting,” in *USENIX Annual Technical Conference*, 2023.

[89] A. Jacobs, M. Gülmez, A. Andries, S. Volckaert, and A. Voulimeneas, “System call interposition without compromise,” in *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2024.

[90] F. Yang, A. Vahldiek-Oberwagner, C.-C. Tsai, K. Kaoudis, and N. Dautenhahn, “Making ’syscall’ a privilege not a right,” *arXiv preprint arXiv:2406.07429*, 2024.