The Astonishing Evolution of Probabilistic Memory Safety

From Basic Heap-Data Attack Detection Toward Fully Survivable Multivariant Execution

André Rösti | University of California, Irvine
Alexios Voulimeneas | TU Delft
Michael Franz | University of California, Irvine

Probabilistic memory safety combines *randomization* and *replication* in the hope that attacks will lead to observable differences across the replicas and hence be detected. It has evolved from simple heap-data protection to full-fledged survivability, harnessing checkpoint/restore facilities and hardware heterogeneity.

As attacks exploiting memory vulnerabilities have become more sophisticated over time, so have the defenses against them. Some of the earliest cyberattacks were focused on directly injecting malicious code into running processes.

Introduction

One of the early successful defenses introduced hardware-based memory page permissions that ensured pages were either writable or executable, but never both, thus stopping direct code injection attacks. Of course, this did not end all cyberattacks; it just led the attackers to come up with more creative approaches, such as return-oriented programming (ROP). Over the course of the past three decades, a vast amount of research has studied ways of enforcing control-flow integrity,[1] the goal of which is to ensure that the execution of a program follows only the paths originally expressed in its source code. This research has significantly raised the bar for attackers trying to divert control flow.

Unfortunately, attackers can cause damage not only through control-flow attacks, but also by changing noncontrol data. Hence, a second body of research has studied more general ways of ensuring memory-data

Digital Object Identifier 10.1109/MSEC.2024.3407648

Copublished by the IEEE Computer and Reliability Societies

integrity. A fundamental reason why enforcing memory-data integrity is hard in general is the existence of pointers and dynamic memory management. There are many things that can go wrong in programs manipulating data via pointers, including *stale pointers* (use-after-frees), in which a pointer access path still remains to a memory area that is no longer occupied by the original data the pointer relates to, and *out of bounds accesses*, in which a valid access path enables manipulation of data it isn't supposed to.

One of the successful approaches to the memory-data integrity problem has been *probabilistic memory safety*, which is the topic of this article. This was originally proposed by Berger and Zorn in their DieHard article.[4] The authors described a system that approximates an infinite-sized heap and in which the memory manager *randomizes* the location of objects. Further, their system can run in *replicated* mode, in which multiple replicas of the data are randomized differently. Since errors or deliberate memory corruption attacks are unlikely to affect the randomized replicas in the same manner, this enables detection of memory corruption errors. This approach combines elements from earlier work on software diversity[2] and *N*-version programming.[3]

In particular, in DieHard the heap is protected probabilistically from overflows, double-frees, use-after-frees, and uninitialized reads. The allocator uses randomized allocations and adds random padding between objects. The additional space between objects mimics an ideal scenario where objects on the heap would be infinitely spaced, and thus buffer overruns would be inconsequential. The random locations of new allocations thwart pointer reuse. Replicated parallel execution of the same program protects from uninitialized reads. Thanks to the randomized addresses produced by DieHard's allocator, uninitialized reads likely yield different random data in multiple executions of the program. DieHard detects these diverging outputs and aborts execution when they occur.

DieHard pioneered probabilistic memory protection for the heap. Other researchers soon seized on the idea of combining randomization and replication more generally. This led to the development of what we today call multivariant execution (MVX) systems, which run multiple *variants* of a program in lockstep and cross-check their states frequently.[5] When such systems detect a divergence, they abort program execution.

MVX systems differ from their precursor, *N*-version systems, in the way program variations are created. In ***N-version*** systems, the assumption is that the different program versions would be created manually by independent teams of programmers (often in different programming languages) from the same specification. Because of the resulting high development costs, such systems have been employed only in niche high-assurance areas, such as fly-by-wire avionics. In contrast to this, in MVX systems, the different *variants* are typically generated automatically by a compiler from a single set of source code files, and the variants are typically more similar to each other. This approach has significantly lower cost because of the automatic variant generation.

As an improvement over the protections afforded by DieHard, in which program variants differ only in their heap layout, many different kinds of software diversity are available for MVX systems, and modern compilers are able to mostly automate the variant creation process. Generally, the more diverse the variants are from each other, the broader the protection. For example, by alternating stack growth direction between variants, the stack can be protected in addition to the heap.

MVX systems have matured significantly since their first conception, broadening their applicability (multithreading, memory-dependent behavior, and GUI applications).[11] However, certain classes of attacks continue to evade detection in most MVX systems. Examples include data-oriented programming attacks and position-independent code reuse.[12,13] The problem in these cases is limited diversity: program variants are not sufficiently different in single-machine setups to cause observable divergences under these attacks. Distributed MVX systems across distinct machines with heterogeneous hardware address the limited diversity problem.[14] The added hardware heterogeneity leads to increased software diversity, improving security. Using a heterogeneous MVX system on processors with diverse instruction set architectures (ISAs) significantly raises the bar for the two aforementioned attacks.

With these advances, MVX systems are already an incredibly powerful defense for a broad class of attacks today. We believe that the two most interesting developments lying ahead for MVX systems are 1) the development of computers with internal heterogeneous hardware that allows these systems to harness hardware diversity in a more efficient manner, and 2) the addition of guaranteed survivability, achieved by combining MVX systems with checkpoint/restore facilities.

As noted, heterogeneous hardware enables the detection of otherwise elusive attacks by adding further diversity into program variants. For example, alignment requirements across different hardware can significantly complicate pointer reuse. Existing systems that provide hardware heterogeneity are set up as a distributed network of separate machines, which adds networking overhead; by colocating heterogeneous hardware within the same computer, this additional overhead can be avoided.

MVX systems can also achieve survivability via checkpoint/restore facilities.[15] This is a natural next

step—recovering detected attacks instead of aborting execution altogether. The straightforward idea is to capture the program state at regular intervals during execution and reverse to a last-known "good" state upon detection of a divergence. In practice, for example, for a webserver application, this may mean dropping an attacker's connection upon an attempted exploit and continuing to serving regular users.

In the remainder of this article, we will revisit the evolution of MVX. We will then cover the interesting aspects and considerations of heterogeneous MVX systems and combine them with checkpoint/restore facilities, based on a prototype system with these capabilities that we have been building. As we explain, such a system can prevent a broad class of memory exploits, without sacrificing availability. We conclude with an outlook toward even more advanced future developments.

The Evolution of MVX

Software Diversity

While, for most purposes, the predictability of deterministic computer programs is a strength, it is detrimental to security: even exploitable "undefined" or unintended behavior is often deterministic. Attackers can therefore prepare for an attack offline, examining and exercising an application until they find a vulnerability. When it is time to deploy the attack on a live system, the predictability of the program execution means that it will likely succeed.

Adding to this problem is the prevalence of monocultures in software. In many domains, a single or only a handful of pieces of software dominates—take, for example, specialized software libraries used by many programs, such as cryptography, compression, or encoding libraries. A single vulnerability in any such popular piece of software may put a large number of users at risk.

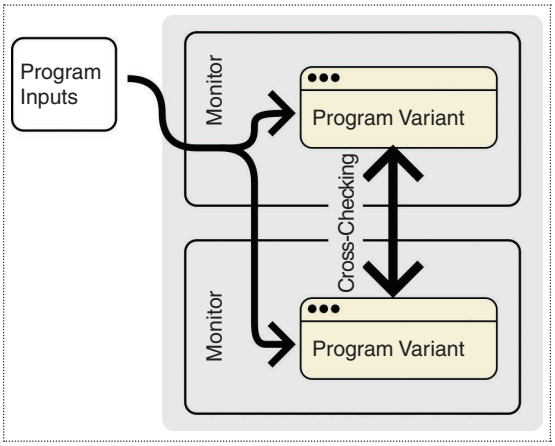

Figure 1. In multivariant execution, automatically diversified program variants execute in parallel on identical inputs while a monitor cross-checks for identical behavior.

To break the high predictability aspects of programs, we need to reintroduce diversity. An early mitigation to do so was Berger and Zorn's DieHard.[4] In DieHard, the memory allocator randomizes placement of objects on the heap, making it much harder to reliably overwrite data in pinpoint locations. These sorts of randomizations of program behavior and implementations are the common theme in (automated) software-diversity-based defenses.

Besides randomizing the heap memory layout, as pioneered by DieHard, many other program aspects can be made nondeterministic. For example, addresses of functions (or even individual basic blocks) can be easily randomized at compile or link time.[2] This added diversity complicates a large number of exploits that depend on the program details being randomized, such as specific expected addresses, offsets, sizes of data structures, the availability and specific structure of gadgets, and so on.

Recently, researchers have explored harnessing hardware platform heterogeneity to produce software variations. Processors implementing various diverse ISAs are available to consumers; programs compiled for these hardwares vary in their used application binary interface (ABI), endianness, and overall memory layout and alignment requirements. Further diversity can stem from the differences in operating system implementations across hardware architectures, such as differences in the system call interface.[10]

No matter how the diversity is obtained, by adding variability, attackers are forced to specialize their attacks to one variant. However, these defenses remain probabilistic—if the attackers "guess" (or somehow learn, through an information disclosure) the right variant, no additional security is provided. Furthermore, probabilistic memory defenses cannot protect against all classes of attacks. For example, code-reuse attacks may succeed even in randomized environments.

MVX

To improve upon the deficiencies of standalone software diversity, MVX systems detect exploits through parallel execution of *multiple* program variants.[6,7] (See Figure 1.) In this way, it becomes impossible for an attack specialized on only one variant to succeed—the attackers have to adapt to all variants at once. For example, code-reuse attacks (such as return-into-libc or ROP) or data-only exploits are still possible even when some software diversity (such as address space layout randomization) is deployed, whereas MVX systems have a much higher probability of detecting them.

Under MVX, all variants receive identical inputs throughout their execution. To an end user, it appears as if only a single program is running. Periodically, a trusted monitor component checks whether all variants are behaving identically. Typically, this "cross-checking" takes place at the granularity of system calls; that is, the monitor verifies that all variants execute the same system calls with equivalent arguments. As the variants are derived from the same source program, their intended benign behavior should be identical. However, we expect a detectable divergence during an attempted attack with high probability because many low-level memory attacks depend on implementation details and undefined behavior, which varies between properly diversified variants. As shown in previous work, the system call behavior does indeed diverge *only* under attack for a large class of exploits. If an attack is detected, all variants are halted immediately in conventional MVX systems.

For certain vulnerabilities, variants can even be constructed such that a divergence is guaranteed. For example, attacks that rely on absolute code addresses can be deterministically spoiled by making executable memory regions disjoint between variants; any executable address is not executable in the other.

We note one limitation that results from using automated diversity: vulnerabilities that are fundamentally a part of an application's design cannot be detected. Automated diversity randomizes only implementation details, keeping the original program semantics intact. If a program's logic is flawed, those flawed semantics are preserved. MVX systems protect primarily against vulnerabilities that are an accidental consequence of implementation specifics (such as memory allocator choices).

DieHard's "replicated mode" qualifies as an early MVX system. The diversity in its variants stems from the use of a memory allocator that randomizes the location of objects on the heap, and DieHard incorporates a monitor that cross-checks the outputs of variants at barriers. This early system was concerned only with reliability, not security.[4] Subsequent systems incorporated more diverse variants by making use of the various sources of automated software diversity outlined previously, such as memory layout and instruction selection.[5,6]

In early MVX systems, the monitor component was implemented in the operating system kernel. The prototype in the work of Bruschi et al.[6] was the first solution to run entirely in userspace. Making the monitor a userspace program is beneficial to security, stability, and ease of implementation. It reduces the trusted code base, honors the principle of least privilege, and makes use of the process isolation provided by the operating system to separate the monitor from the monitored application. However, this approach can increase the runtime overheads significantly. VARAN[8] aimed to improve performance through binary rewriting; in this system, the monitor resides in the same address space as the application, and all system call instructions are rewritten to be jumps into the monitor. In contrast to other MVX systems, such as Orchestra,[7] whose goal is security, VARAN's aim is to increase reliability in case of crashes. VARAN cannot provide any security guarantees because of its architecture; the monitor is as vulnerable as the application it aims to protect. On the other hand, ReMon,[9] a subsequently published system, aims to be both secure and performant by using a hybrid monitor design and specific cross-checking policies. The basic idea is to categorize system calls as security critical and nonsecurity critical. Then, a hardened in-process monitor efficiently handles nonsecurity-critical system calls, while a slower cross-process monitor handles the rest.

Previous research has explored how to overcome some of the limitations of MVX systems, such as working around the false positive divergences that would be observed because of scheduling differences in multithreaded programs.[11] Shared memory support, support for address-dependent behavior, and signal delivery issues have also been discussed.

Baseline: A Conventional MVX System

MVX systems are traditionally implemented on a single physical machine. Let us first dive into the typical design considerations of such a "conventional" MVX system, before we look into extending it to a heterogeneous, multimachine environment.

A typical MVX system comprises at least the following three functionalities:

- A scheme for obtaining multiple distinct *program variants*. The variances between variants are crucial for determining which types of attacks the MVX system will be able to defend against.
- A *cross-checking* component, which compares the current execution state of all variants, assuring continued congruence of each variant's state. Upon a divergence, typical MVX systems abort execution as an attempted exploitation of a vulnerability is suspected.
- A *replication and multiplexing* component, which makes the MVX transparent to the end user. The goal is to make it appear as though only one program were running, by sharing necessary resources that cannot be duplicated.

Variant generation can take place at any stage of the program lifecycle. The cross-checking and replication

take place at runtime and are responsibilities of a "monitor" component.

Variant Generation

Any of the approaches established in the large body of research into software diversity can be used to generate variants for MVX execution.

A fairly recent, promising idea for variant generation is the use of diverse hardware, particularly processors with differing ISAs.[14] ISAs differ in their available instructions, instruction encodings, and sizes, leading to entirely different binaries after compilation. Additionally, ABIs and (often) system call interfaces vary across architectures. Thus, if the source code of the target application is available, compiling and running multiple variants on different ISA hardware protects against all vulnerabilities that expose any difference across these various layers. For example, consider a code injection attack that injects the same binary payload into all variants. The only way for such an attack to succeed is if the payload decodes to valid instructions in *both* ISAs simultaneously; even a single ill-formed instruction will cause an exception in one of the variants that can be detected.

Monitor Component

The monitor component ensures lockstep execution of the program variants by synchronizing and cross-checking all program variants at "rendez-vous points." Additionally, the monitor replicates the same program inputs to all variants. A straightforward choice for rendez-vous points is to use system call invocations: this provides sufficient security guarantees since any dangerous outside-world interaction that a program variant can make is through the use of a system call. However, denial-of-service attacks are not detectable at the system call level since attackers can stall a program without the use of any system calls. Such attacks can be detected by adding rendez-vous points at regular time intervals.

System call interposition and cross-checking. To cross-check variants, a means of intercepting system calls efficiently is needed. Naively implemented, system call interposition can end up being the bottleneck of the system as it requires switching from a variant execution context to the monitor's context. It is critical that the monitor is sufficiently isolated from the variant; a vulnerability in the variant should remain contained to the variant. Without proper isolation, attackers could compromise the monitor itself and circumvent its checks.

System call interposition and, more broadly, monitor implementations, can be categorized into three categories: in-kernel monitors, which perform all their work inside the kernel; cross-process monitors, which intercept system calls from a userspace process separate from the variant process; and in-process monitors, which are colocated in the same process as the variant.

In-kernel monitors hook into the system call entry and exit handlers, and they have full access to the variant's memory, allowing them to freely inspect and modify any data needed. However, the principle of least privilege suggests that incorporating a large piece of software like a monitor component in the kernel, when only a small portion of that software requires escalated privileges, is unwise. Compared to userspace development, writing kernel code is more involved, and bugs are more consequential.

In cross-process monitors, a separate userspace monitor process receives a notification from the kernel whenever the variant process attempts to issue a system call or receives a signal. Debugging features of the operating system ("ptrace" in Linux) enable this. Reading the variant's memory requires an indirection through the kernel (for example, "process_vm_readv"). These context and mode switches properly isolate the monitor from the variant, but they come with hefty performance overheads.

In-process monitors, like cross-process monitors, are implemented in userspace. Unlike cross-process monitors, they are part of the same address space as their monitored application. Implemented as a shared library, for example, the monitor code can be "injected" into the variant process (using "LD_PRELOAD" in Linux). An alternative is to instrument code using binary rewriting. Such approaches allow fast variant memory reads from the monitor. However, the monitor is not isolated from the variant whatsoever in these approaches, limiting their use to scenarios where reliability, but not security, is a concern.

We believe an ideal compromise for security-focused systems is a hybrid in-kernel and in-process approach: locating the monitor within the monitored process allows low-overhead interactions, while a minimal kernel module provides the necessary isolation. This strikes a balance between the performance penalty of a fully isolated monitor in a separate process and the complexity and exposure of a fully in-kernel implementation.

Once a system call has been intercepted, the monitor cross-checks it between all variants. This encompasses comparing the system call number and its arguments to assure all variants are requesting equivalent services from the operating system. Note that, at this stage, system call arguments must be serialized by the monitor for comparison. Pointer arguments, for example, must be resolved to their pointed-to values as the addresses in each variant might diverge. (In fact, in properly diversified variants, addresses *should* diverge.) When

pointers point to variable-length buffers, other system call arguments can be inspected to learn the length of the pointed-to buffer. For example, the "read" and "write" system calls take pointers to preallocated buffers to read data into or write data from; an additional argument gives their (maximum) length.

Input–output replication and multiplexing. To a user of an application, MVX should be indistinguishable from running a single program variant. This means that program inputs given by the user must be replicated to all program variants. Conversely, although all program variants produce some output, only one result should be returned to the user. To facilitate this, MVX systems incorporate input–output (I/O) replication functionalities. Any outward-facing resources, such as open sockets and standard I/O, are shared by all program variants. System calls interacting with these outward-facing resources are executed once—if all variants executed them individually, it would be observable to the user (for example, as multiple outputs) or lead to divergences in the variant's behaviors.

Most MVX systems implement a leader/follower design to facilitate replication. A designated leader variant performs all outward-facing I/O and informs its followers of the results upon completion. The followers copy results back to make it appear as though the system call executed locally.

Additionally, certain resource identifiers, such as file descriptor numbers and process IDs, often diverge between variants, even though the referenced resources are the same. To facilitate comparison between variants without false positive divergences, the monitor should multiplex resource identifiers. One approach is to keep a set of "canonical" identifiers and a mapping to their local values.

The Cost of Conventional MVX Systems

Since MVX systems require the parallel execution of N program variants, one could reasonably expect at least an N-times overhead. Surprisingly, the costs of deploying such systems are often lower in practice.

Although *computation* happens N-fold in an MVX system, *I/O* is shared between variants. This helps I/O-bound applications, such as webservers. The first MVX system reported ≈16% overhead for an unloaded Apache server (two variants), although that number did rise to ≈1.9× in a saturated network.[5] Later work was able to reduce this to ≈34%.[9] Throughput was measured from a client located within the same rack as the server. If the client has a higher-latency connection—a more realistic scenario—the overheads reduce to less than 2.5%.

In theory, MVX systems will introduce N-fold runtime overheads for applications that are already under full load during regular (non-MVX) execution. In practice, CPU cores are often underutilized, and MVXs may be specifically deployed on systems with ample spare resources; therefore, the reported overheads for benchmarks such as SPECint2006 are between 18% and 4%. In the best case, MVX systems have achieved near-native performance.[8,9]

Adding Heterogeneity: MVX Across Diverse Machines

Running all variants on the same physical machine, as is done in the vast majority of established MVX systems, leaves programs vulnerable to architecture-specific exploits. Diversity obtained through heterogeneous hardware is powerful because it further reduces the set of undetectable vulnerabilities. It is relatively easy to obtain additional protections, stemming from the ABI and ISA heterogeneity, just by recompiling the same software for different platforms. Heterogeneity can also come from other hardware or convention differences, such as different endianness, structure layouts, or available system calls.

In current heterogeneous MVX systems, monitors run on physically separate machines; this is called "distributed MVX." These machines are interconnected with a high-bandwidth, low-latency link, and this link is utilized to cross-check and replicate results between variants.

As opposed to a system that executes variants on the same machine, a heterogeneous and distributed MVX environment faces additional challenges. For one, cross-checking between variants involves network communication, significantly shifting the overheads. Cross-checking the states of the variants becomes more involved as the state representations differ across architectures. For example, the set of available system calls, and their numbers, differ on *x86_64* and *ARM64* Linux. A *canonicalization* phase, which translates an architecture-specific state into a platform-agnostic canonical form, is required.

Networking Considerations

In a local MVX system, shared memory primitives can facilitate fast cross-checking and replication across variant processes. In a distributed system, the network communication quickly becomes the biggest overhead. To achieve a workable system, a low-latency, high-bandwidth connection between constituent machines is paramount. In previous work, we have employed InfiniBand connections between directly connected hosts in the same rack with remote direct memory access network drivers to minimize these overheads. Further optimizations to consider include the following:

- *Cross-checking policy*: Relieving some noncritical system calls from cross-checking, or deferring cross-checking to a later point, reduces the amount of required network communication and opens the door for the optimizations discussed here.
- *Asynchronous replication*: For cross-check-exempted system calls, the leader does not need to execute strictly in lockstep with followers; asynchronous replication makes use of this.
- *Batching of replication information*: Perhaps counterintuitively, batching replication information can *reduce* overall system latency.

Cross-checking policy. It has been shown that, depending on the class of attacks one wishes to defend against, cross-checking all system calls is not required. An efficient monitor should support different policies that omit certain system calls from cross-checking to adjust a safety/overhead tradeoff based on the application. Previous work[9] used the following policies:

- *Code execution*: checks only *execve*, *mmap*, and *mprotect*
- *Information disclosure*: checks all read/write system calls in addition to the ones checked by the *code execution* policy
- *Comprehensive*: cross-checks all system calls.

Developers of MVX systems manually define these policies ahead of deployment. A policy can be as simple as a list of system calls deemed safe under the assumed threat model. More detailed policies may additionally consider arguments. For example, if the goal is to protect against information disclosure to a remote attacker, one may consider write system calls to be safe unless they write to a socket. Defining policies is a simple one-time process. During execution, the policies are applied automatically.

Asynchronous replication. I/O replication is required for system calls interacting with outward-facing resources. In a naive implementation, the leader node, after sending replication information, may wait for the follower(s) to acknowledge receipt.

However, this is not necessary for system calls that are exempt from cross-checking. Instead, the leader can append its replication information to a queue and immediately continue variant execution, while another process disperses the replication information in parallel. Through this "fire-and-forget" approach, some of the network communication happens concurrently with useful variant execution on the leader. Once the followers arrive at the same point in program execution, they will consume the required replication information from the queue and continue. If they arrive at that point before the leader does, they block and wait for the leader.

Replication batching. In addition to the cost of exchanging useful information, each network exchange between monitors adds some amount of communication overhead, such as required message headers and acknowledgments. This overhead can become unreasonably large and dominate over the useful payload exchanged, especially when a large number of small messages is exchanged. For example, the "gettime" system call must be replicated (all variants should receive the same time stamp), but the data sent are only a few bytes (a long integer, the time stamp).

Perhaps counterintuitively, batching replication information can reduce the overall latency of the system. The general idea is for the leader to "collect" replication information for multiple system calls until the replication batch reaches a certain threshold size or until the system encounters a system call that requires synchronization. The collected information for multiple system calls is then sent, all at once, to followers. In this way, a single message header and acknowledgment will carry a payload of multiple system calls, amortizing the communication overhead. Since program execution is considerably faster than network communication, pausing variants until a batch of replication information is received and then executing a sequence of system calls "all at once" overall benefits the system performance.

Canonical System Call Form

In the cross-checking phase, MVX monitors must ensure the same system calls are issued with identical arguments. On a conventional MVX system, this consists of a straightforward comparison of the attempted system call number and its serialized arguments.

On a heterogeneous system, cross-checking requires an additional "canonicalization" phase. In this phase, data are transformed into a platform-agnostic normalized form. System call interfaces, for example, argument data types, diverge between different architectures. For example, if the heterogeneous system employs one little-endian and one big-endian architecture, all integer arguments larger than one byte must be translated into a canonical form before comparison.

For results replication, a de-canonicalization process also needs to take place. For example, the "gettime" system call on *x86_64*, which is not present on *ARM64* Linux, may be canonicalized to a "gettimeofday" system call upon entry for cross-checking. If the call is executed on *x86_64*, the time stamp return value obtained must be plugged into a "struct timeval", the expected return

value of "gettimeofday" on *ARM64*, and vice versa if the "gettimeofday" call is executed on ARM, and its results must be replicated into a "gettime" return value.

Adding Survivability: Checkpoint/Restore

We have discussed MVX behavior during regular, benign execution. What should an MVX system do when, during cross-checking, a divergence is detected? Previous research answered this question by stopping execution of the program altogether. We believe an MVX system could be made more useful by managing divergences using checkpoint/restore. MVX systems are often deployed to protect critical infrastructure, such as webservers; in such applications, availability is an important requirement that is usually as important as security. By providing MVX systems with a checkpoint/restore mechanism, we can satisfy both of those requirements: not just security, but also availability.

Checkpoint/restore has been an established technique since the 1990s to enable fault tolerance.[15] The program state, such as the memory and register contents, is captured and stored at regular intervals. When needed, this stored state can be restored to resume program execution at the previously stored "snapshot."

Integrating MVX with checkpoint/restore is a straightforward idea: the MVX monitor takes on an additional responsibility of creating checkpoints at regular intervals during regular program execution. (See Figure 2.) These checkpoints are created only as long as all variants agree on the program state. Upon a divergence, the last checkpoint can be restored to resume execution from a previous uncompromised state. Restoring to a previous checkpoint may discard some useful work that the application already did (for example, in the case of a webserver, dropping some users' connections). However, the overall downtime is less than if the application were terminated and restarted because checkpoints keep most of the built-up state of the application.

Checkpointing Locations

One critical consideration when adding checkpoint/restore functionality to an MVX system is the choice of checkpointing location and frequency. The checkpointing location should be chosen such that the target application does not become "confused" by potentially changed external circumstances. For example, in a webserver application, the middle of a connection-handling routine would not be a good place to create a checkpoint since that connection may no longer exist when the checkpoint is restored. Instead, checkpoints should be created at a "logical" beginning of a unit of processing. Most long-running applications are structured around a main loop that processes events. The start of an iteration of such a main loop is usually a good checkpointing location.

An MVX system may leave the definition of the checkpointing location to the system administrator, or future work may investigate how to automatically determine suitable checkpointing locations.

Another consideration is checkpointing frequency/period: Assume checkpoints are created every *n*th time execution passes a potential checkpoint creation location. Choosing the period *n* decides a tradeoff between benign program performance and performance under attack. With a lower frequency (a larger *n*), the overhead of checkpoint creation is minimized during regular execution, but the execution takes a "bigger step back" upon checkpoint restoration, so performance under attack decreases. Conversely, a low *n* causes larger overheads even when no checkpoints are needed but reduces the disruption when restoring is necessary.

Creating Checkpoints

At the process level, a checkpoint as a minimum will capture the memory and register contents. This will allow purely functional programs to be restored at a later time. However, almost all useful programs interact with the operating system in some way, meaning the operating system state associated with a process must be captured as well. This includes information such as contents of opened files, opened external sockets, child processes, etc. Checkpoint/Restore in Userspace is a utility that is able to capture all of this state for Linux processes and store it to disk. It does so by pausing the process and attaching to it using *ptrace*.

For many purposes, such a complete checkpoint is not required. For example, as long as a webserver application has robust error-handling code, it should have no issue handling a suddenly dropped

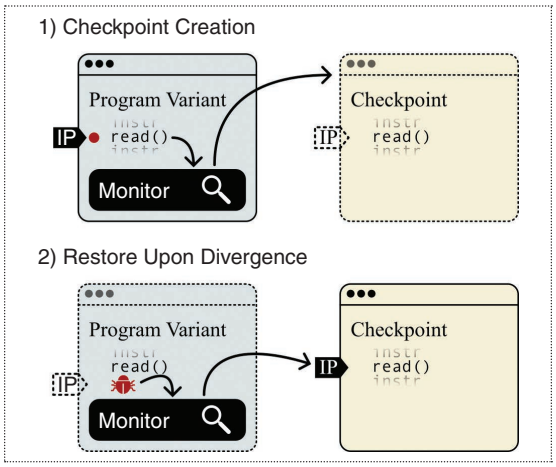

Figure 2. At regular intervals, we create a checkpoint of each variant's execution state. If a divergence later occurs, we can restore the most recently created checkpoint. IP: instruction pointer.

connection. There is, therefore, a tradeoff between completeness of the checkpoint and time taken to create the checkpoint.

Alternatively to on-disk checkpoints, checkpoints for MVX applications can be created as in-memory only as they are needed on a much shorter time scale than other checkpoint/restore algorithms. Typically, these checkpoints will live only for a few seconds before being supplanted by the next captured checkpoint during regular execution. When creating an in-memory checkpoint, copy-on-write semantics can be used to further reduce the overhead: instead of copying all memory pages of the variant, one marks them all as read-only and only copies those pages that the process attempts to write to.[15] On Linux, the "fork" system call provides this functionality.

Outlook

Combining MVX systems with hardware heterogeneity and checkpoint/restore facilities results in unavoidable additional runtime overheads. However, in many domains, the superlative security and survivability assurances that such systems enable often justify the lower throughput. Critical infrastructure systems (such as airline reservations, power, water, traffic, and banking) often comprise long-running applications that cannot easily be restarted and where security is of critical importance. Such systems could greatly benefit from execution using an MVX approach that allows continued execution under attack.

We are currently evaluating a combined checkpoint/restore–MVX system that we have been building. Our system prototype runs on a distributed network of heterogeneous hardware to harness the security benefits achieved by ISA diversity. The system we have built is stable for real-world webserver and database applications and will be the subject of a future publication.

In current heterogeneous MVX systems, network communication undoubtedly is the largest contributor to overheads—even in the face of the discussed optimizations. Although the achievable overheads are acceptable for high-stakes domains, such systems are less appealing for applications that cannot tolerate these slowdowns. Achieving heterogeneity *locally*, that is, on one physical machine, could be a promising next step for opening the doors to using such systems more broadly. Performance has been the primary focus of hardware developers in the past. However, we believe that allocating more real estate on processors to security, for example, through heterogeneous ISA processor architectures, could have real benefits worth investigating. For example, novel designs might implement different ISAs on each core of a processor while continuing to share the same memory. This would enable the construction of heterogeneous MVX systems with architectural vulnerability resilience at much lower overheads. ■

Acknowledgment

We thank Stijn Volckaert and Jonas Vinck, Adrian Dabrowski, and Felicitas Hetzelt for helpful discussions.

This research was funded in part by the Office of Naval Research (ONR) under grant N00014-21-1-2409; any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of ONR. André Rösti is the corresponding author.

References

1. M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proc. 12th ACM Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA: Association for Computing Machinery, 2005, pp. 340–353, doi: 10.1145/1102120.1102165.
2. P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proc. IEEE Symp. Secur. Privacy*, Berkeley, CA, USA, 2014, pp. 276–291, doi: 10.1109/SP.2014.25.
3. A. Avizienis, "The N-version approach to fault-tolerant software," *IEEE Trans. Softw. Eng.*, vol. SE-11, no. 12, pp. 1491–1501, Dec. 1985, doi: 10.1109/TSE.1985.231893.
4. E. D. Berger and B. G. Zorn, "DieHard: Probabilistic memory safety for unsafe languages," in *Proc. 27th ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI)*, New York, NY, USA: Association for Computing Machinery, 2006, pp. 158–168, doi: 10.1145/1133981.1134000.
5. B. Cox et al., "N-variant systems: A secretless framework for security through diversity," in *Proc. 15th Conf. USENIX Secur. Symp. - Volume 15 (USENIX-SS)*, Berkeley, CA, USA: USENIX Association, 2006, Art. no. 9.
6. D. Bruschi, L. Cavallaro, and A. Lanzi, "Diversified process replicæ for defeating memory error exploits," in *Proc. IEEE Int. Perform., Comput. Commun. Conf.*, Piscataway, NJ, USA: IEEE, 2007, doi: 10.1109/PCCC.2007.358924.
7. B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space," in *Proc. 4th ACM Eur. Conf. Comput. Syst. (EuroSys)*, New York, NY, USA: Association for Computing Machinery, 2009, pp. 33–46, doi: 10.1145/1519065.1519071.
8. P. Hosek and C. Cadar, "VARAN the unbelievable: An efficient N-version Execution framework," in *Proc. 20th Int. Conf. Archit. Support for Program. Lang. Oper. Syst. (ASPLOS)*, New York, NY, USA: Association for Computing Machinery, 2015, pp. 339–353, doi: 10.1145/2694344.2694390.

9. S. Volckaert et al., "Secure and efficient application monitoring and replication," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf. (USENIX ATC)*, Berkeley, CA, USA: USENIX Association, 2016, pp. 167–179.
10. A. Venkat et al., "HIPStR: Heterogeneous-isa program state relocation," in *Proc. 21st Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2016, pp. 727–741, doi: 10.1145/2872362.2872408.
11. S. Volckaert, B. Coppens, B. De Sutter, K. De Bosschere, P. Larsen, and M. Franz, "Taming parallelism in a multi-variant execution environment," in *Proc. 12th Eur. Conf. Comput. Syst. (EuroSys)*, New York, NY, USA: Association for Computing Machinery, 2017, pp. 270–285, doi: 10.1145/3064176.3064178.
12. 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 *Proc. IEEE Symp. Secur. Privacy (SP)*, San Jose, CA, USA, 2016, pp. 969–986, doi: 10.1109/SP.2016.62.
13. E. Göktas et al., "Position-independent code reuse: On the effectiveness of ASLR in the absence of information disclosure," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, London, U.K., 2018, pp. 227–242, doi: 10.1109/EuroSP.2018.00024.
14. A. Voulimeneas et al., "Distributed heterogeneous N-variant execution," in *Proc. 17th Int. Conf. Detection Intrusions Malware, Vulnerability Assessment (DIMVA)*, Lisbon, Portugal. Berlin, Germany: Springer-Verlag, Jun. 24–26, 2020, pp. 217–237, doi: 10.1007/978-3-030-52683-2_11.
15. J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 10, pp. 972–986, Oct. 1998, doi: 10.1109/71.730527.

André Rösti is a Ph.D. student at the University of California, Irvine, Irvine, CA 92697-3425 USA. His research focuses primarily on systems software. Rösti received his M.S. in computer science from the University of California, Irvine. Contact him at aroesti@uci.edu.

Alexios Voulimeneas is an assistant professor at TU Delft, 2628 XE Delft, The Netherlands. His research interests broadly include systems security, operating systems, and computer networks. Voulimeneas received his Ph.D. in computer science from the University of California, Irvine. Contact him at a.voulimeneas@tudelft.nl.

Michael Franz is a distinguished professor at the University of California, Irvine, CA 92697-3425 USA. His research focuses on computer security, systems, and software. Franz received a doctor of science from ETH Zurich. He is a Fellow of IEEE, the Association for the Advancement of Science, the Association for Computing Machinery, and the International Federation for Information Processing. Contact him at franz@uci.edu.