sMVX: Multi-Variant Execution on Selected Code Paths

SengMing Yeoh
sengming@vt.edu
Virginia Tech

Xiaoguang Wang
xgwang9@uic.edu
University of Illinois Chicago

Jae-Won Jang
jjang3@vt.edu
Virginia Tech

Binoy Ravindran
binoy@vt.edu
Virginia Tech

Abstract

Multi-Variant Execution (MVX) is an effective way to detect memory corruption vulnerabilities, intrusions, or live software updates. A traditional MVX system concurrently runs multiple copies of functionally identical, layout-different program variants. Therefore, a typical memory corruption attack that forges pointers can succeed on at most one variant, leading the other variant(s) to crash. The replicated execution adds software security and reliability but also brings multiple times of CPU and memory usage.

This paper presents sMVX, a flexible multi-variant execution system *replicating variants only on the selected code paths*. sMVX allows end-users to annotate a target program and indicate sensitive code regions for multi-variant execution. Such code regions can be authentication-related code or sensitive functions that handle potentially malicious input data. An sMVX runtime only replicates the sensitive functions and executes them in lockstep. We have implemented a prototype of sMVX using an in-process code monitor. The sMVX monitor supports the selected code paths MVX from within the target program's address space, but the monitor is isolated from the target's code by the Intel Memory Protection Keys (MPK). We evaluated the sMVX using a benchmark suite and two server applications. The evaluation demonstrates that sMVX exhibits a comparable performance overhead to state-of-the-art MVX systems but requires 20% fewer CPU cycles and 49% less memory consumption on server applications.

CCS Concepts: • **Security and privacy → Software and application security**; **Systems security**.

Keywords: Multi-Variant Execution, Memory Protection, Software Security

ACM Reference Format:
SengMing Yeoh, Xiaoguang Wang, Jae-Won Jang, and Binoy Ravindran. 2024. sMVX: Multi-Variant Execution on Selected Code Paths. In *25th International Middleware Conference (Middleware '24), December 2–6, 2024, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3652892.3654794

1 Introduction

The N-Version system, or the Multi-Variant Execution (MVX), is a method to improve software security and reliability by running functionally equivalent program variants concurrently [7, 9]. An MVX monitor dispatches the input data to all program variants at runtime and periodically checks the execution results. MVX has been used in many scenarios, such as seamless software updates [16, 37, 57] and intrusion detection [39, 47, 53]. These approaches leverage the replicated program variant to replace an old software instance or detect an attack. The additional replication and redundancy bring reliability to the software deployment.

Recently, security researchers proposed to use MVX to detect memory corruption attacks [25, 39, 40, 47, 51, 53, 59]. These MVX systems generate program variants of the same functionality but are different in-memory layouts. For example, existing approaches used non-overlapping address spaces [25, 39, 59], reversed stack order [40], or even heterogeneous instruction set architectures (ISAs) [50, 51, 53] to generate program variants. It relies on the fact that an exploit can only target one particular program variant (*e.g.,* an address space layout or a particular architecture), while the same exploit payload will cause other variants to crash or have inconsistent execution results [25, 39, 47].

Although existing research efforts have shown MVX is effective in detecting bugs in server applications [9, 25, 39, 46, 47], operating system kernels [6, 59] and even micro-architecture behaviors [50, 51, 53], there are a few key issues in existing MVX design. First, the MVX systems require *several program instances running concurrently*, which wastes CPU cycles and increases memory usage. Second, the whole program replication makes variant synchronization complicated. In particular, the MVX monitor often simulates system calls (or library calls) for the replicated variants to avoid re-executing I/O requests. A larger execution trace requires more system calls to be emulated. Furthermore, handling *process/thread creation* in *initialization code* is challenging for MVX variant synchronization [48]. Replication of such code regions does not bring significant security benefits.

In this paper, we propose sMVX, a new mechanism of MVX systems that *replicates the variant execution only on selected code paths*. Thus, it reduces the number of unnecessary instructions executed due to the replications of the whole program execution. We further applied the sMVX mechanism to selectively replicate and protect sensitive code, such as authentication-related code or functions that interact with the (potentially) malicious external input data, to improve software security. Although sMVX requires end-users to annotate sensitive code regions, we also developed a tool to identify sensitive code paths semi-automatically. The tool was built on a dynamic taint analysis framework – `libdft` [23] and would generate a list of sensitive functions. Application programmers can annotate the code regions for multi-variant execution using their knowledge of the program logic or leveraging the dynamic analysis results. At runtime, the sMVX monitor will create program variants containing only sensitive functions.

To showcase this idea, we implemented sMVX with an in-process code monitor. The code monitor runs within the same address space of the target process but is isolated by the Intel memory protection

Middleware '24, December 2–6, 2024, Hong Kong, Hong Kong

key (MPK) [28]. We demonstrate the usability of sMVX by running two server applications and a benchmark suite under sMVX. The evaluation shows that sMVX performs similarly to a state-of-the-art high-performance MVX monitor – ReMon [49]. However, sMVX allows fewer code paths to be replicated.

Overall, we make the following contributions:

- We identified the code and memory replication issue in traditional MVX systems and proposed a new MVX mechanism that only replicates a set of user-defined sensitive function paths to reduce the CPU and memory usage.
- We present the design of sMVX for sensitive code path identification, instrumentation, and variant generation. We also describe the sMVX implementation with an in-process code monitor protected by the Intel MPK; sMVX is open-sourced: https://github.com/ssrg-vt/sMVX.
- We report the evaluation results of sMVX, discuss the lessons learned and outline potential future improvements in this direction.

2 Background and Threat Model

2.1 Background

Multi-Variant Execution. The core concept behind the multi-variant execution is to secure programs by replication. In particular, by having multiple diversified program variants, MVX systems ensure attack payloads will only work against a subset of the variants, and *any divergence in the flow of execution between variants* will be picked up by a monitoring agent. The diversification can be in the form of stack growth direction [40], non-overlapping memory layout [9, 25, 39, 47, 59], different compiler options [32], or even distinct processor architectures [51, 53]. The MVX monitor typically uses the *lock-step check* (on system calls or libc calls) to periodically verify that all variants are still in the same (correct) state.

Besides detecting the execution divergence, the MVX monitor also ensures all variants execute under the same input. This is important for running multiple program variants on a single machine, as all variants have to interact with the same operating system (OS) I/O buffer [9, 25, 51, 53]. Allowing all variants to access the same I/O buffer will lead to unpredictable execution results and even program crashes. To solve this issue, most MVX systems only allow *a leader variant* to access the system I/O buffer and simulate the I/O operations for *follower variants* when the OS kernel gives back control to the application code. For example, an MVX monitor can emulate a file read by copying the corresponding file content to the follower variant's memory buffer and returning the data size to the application code [9, 25, 39, 47, 51, 53, 59]. After that, the follower variant continues to execute as if it has accessed the file.

The I/O emulation mentioned above requires the MVX monitor to intercept these I/O operations while still keeping the monitor isolated from the target process. Without isolation, a (potentially) malicious program may bypass the MVX monitor's I/O interception and emulation, thus hiding the malicious behavior. Based on the monitor's isolation technique, there are generally two kinds of MVX monitors – the cross-process monitors and the in-process monitors [8, 49]. The former one typically uses a `ptrace` process to host the monitor code for better isolation while keeping the target process in the `ptrace` child process [39, 51]. The monitor and the target have strong inter-process isolation, but the monitor also suffers from higher performance overhead as each I/O interception requires four context switches[1].

Other approaches place the MVX monitor inside the target process but within a higher privilege domain [9, 25, 53, 59], such as hosting the monitor inside an operating system kernel [9, 53, 59] or a hypervisor [25]. Although these approaches achieved strong monitor execution protection, replicating multiple program variants utilizing numerous OSes or machines significantly increases the computing resources. This paper explores an alternative perspective on enhancing the MVX system, seeking to address the question: *Can we minimize resource consumption in MVX systems while upholding a comparable level of security guarantee*?

Intra-Process Monitor Isolation and Memory Protection Keys. Another critical issue for MVX systems is the MVX monitor's placement. As we mentioned earlier, the MVX monitor needs to access (hook) the target application's system calls or library calls, but the monitor must also be isolated from the target's code. Intra-process isolation is an approach to compartmentalize software components within the same address space. There are a few recent research efforts aimed to create compartments for intra-process isolation [26, 29, 30, 54, 58]. For example, light-weight context (lwC) modifies the OS kernel to provide independent isolation units within a process [29]. Following this direction, recent work leverage Intel MPK to achieve sensitive data isolation with cheaper performance costs [15, 36, 45].

Memory Protection Keys for Userspace (PKU) was introduced as an extension of the memory management architecture in Intel Xeon Scalable family (a.k.a., Skylake-SP) [10]. It provides a mechanism to enforce page-granularity protection without modifying the page tables, even when an application updates the protection permission (PKEY). Therefore, it does not require TLB shoot downs and subsequent TLB misses. With MPK, bits 62:59 of each page table entry can be associated with one of the 16 available keys (PKEY). A new 32-bit thread-private protection key rights register for user pages (PKRU) was introduced to store the permissions of the 16 keys. For each key, there are two bits in the PKRU indicating the permissions for the thread currently running on that core: *write disabled* and *access disabled*. To set/change permissions of a memory domain, an unprivileged instruction `wrpkru` can be used to update the PKRU register. The memory permissions can be updated instantly. The protection key only works for *data memory accesses*. Interestingly, if the code pages are associated with an *access disabled* protection key, the code will be executable only (a.k.a., *the execute-only memory (XoM)* [1]). sMVX hides the MVX monitor with code location randomization and leverages XoM to prevent trampoline code from leaking out the monitor location.

2.2 Assumptions and Threat Model

We assume the application source code is available to sMVX users to instrument the selective protection regions and analyze pointer alias. We assume the attacker has access to the target binaries, such as the application code, its shared libraries, and the monitor. At runtime, the attacker can only access the target process remotely through the standard I/O interface, namely a socket connection. The attacker can send arbitrary data to the target process. Similar to most existing MVX systems [25, 39, 46], sMVX does not bring

[1]Intercepting a system call executed in the target process requires two user/kernel context switches for the target process and two context switches for the monitor [49].

any exploit mitigations but relies on the diversified variants execution to detect the potential exploit. We assume a strong trusted computing base (TCB), including the OS kernel and the compiler toolchain. Side-channel attacks and kernel vulnerability exploits and mitigations are out of the scope of this paper.

3 Design and Implementation

3.1 Overview

sMVX aims to reduce the resource usage of the MVX system by only replicating the sensitive code paths. In our current design, the user is required to instrument the code region that contains the sensitive functions (*i.e.,* the sMVX protected region). An sMVX runtime monitor loads the target program binary and generates a variant of the target program *only using the sensitive functions* (Figure 1 (c)). The replicated variant and the main variant do not share any overlapping addresses in their address spaces (Section 3.2). Therefore, an exploit targeting one of the variants will likely be captured by the other because of the potential memory address errors. Figure 1 further illustrates a comparison of application execution under sMVX (Figure 1 (c)) with traditional MVX systems (Figure 1 (b)) and the standard application execution (Figure 1 (a)).

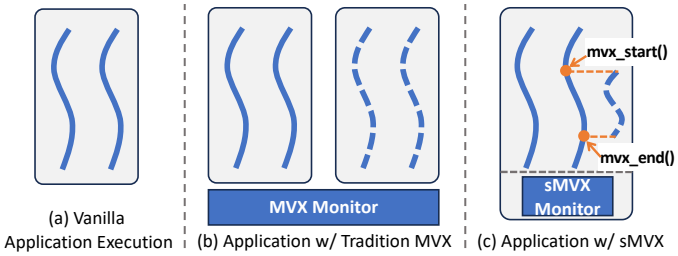

Figure 1. An example of selective replicated execution only for potentially malicious input handling.

Within the protected region, the sMVX monitor performs libc call synchronization and checking through a secure IPC channel (Section 3.3). By doing this, we reduce the context-switches overhead in a traditional `ptrace`-based MVX monitor [39, 51]. The sMVX monitor is placed inside the target address space, but the monitor's memory pages are isolated from the target program by the Intel MPK (Section 3.4). Keeping the monitor within the process makes the sMVX lightweight. Note that sMVX can also protect arbitrary regions of code and is not solely restricted to protecting security-sensitive areas. Extending the protection region to the entire program execution lifetime (*e.g.,* protecting the `main()` function) will be similar to the traditional whole-process MVX systems.

3.2 sMVX Protected Region Instrumentation

To set up sMVX, end-users have to annotate code regions that contain the sensitive functions. sMVX provides a library API of auxiliary functions (*e.g.,* `mvx_*()` in Listing 1) that define the protected region and cooperate with the sMVX runtime for variant creation and termination. The protected code region can be well-known attack surfaces, such as the worker thread in a server application or functions interacting with external I/O data.

As shown in Listing 1, applications are required to call the `mvx_init()` function first. The `mvx_init()` sets up the protected memory regions and load the sMVX monitor. It also initializes the protection keys (*PKEYs*) and associates the *PKEYs* with the corresponding memory pages. The `mvx_start()` and `mvx_end()` are a pair of functions that defines the protected region. The `mvx_start()` function triggers the process's cloning, updating shared values, and creating the shared memory IPC for library calls simulation. It further makes the follower variant jump to the desired address. The `mvx_start()` takes the protected function name, the number of arguments, and the argument variables of the protected function as its arguments (Line 4 in Listing 1). The `mvx_start()` resolves the sensitive function address from the provided function name argument, prepares the thread context for the follower variant, and redirects the control flow to start the follower variant execution.

```
1 int main(void) {
2     mvx_init();
3     /* Unprotected area */
4     mvx_start("protected_func", 2, arg1, arg2);
5     protected_func(arg1, arg2);
6     mvx_end();
7     /* Unprotected area */
8     return 0;
9 }
```

Listing 1. Example of a function protected by sMVX

The `mvx_end()` function is called at the end of the protected code regions. The major role of `mvx_end()` is to merge the variants execution into a single program execution context. Specifically, it synchronizes the lock-step checks and waits for checks to complete before allowing the leader variant to continue. Additionally, it may trigger an alarm if the execution outcomes of the variants diverge, signaling a potential attack. To accomplish this, the `mvx_end()` function utilizes the `wait()` system call, effectively pausing to allow the follower variant's thread to finish its execution.

It's worth emphasizing that there's no necessity to synchronize memory back from the variant to the original function upon the variant's termination, as the variant exclusively serves the purpose of detecting inconsistent execution behaviors. Furthermore, it's important to highlight that the `mvx_start()` and `mvx_end()` sMVX function pair can be invoked multiple times within a program, accommodating the protection of numerous code regions. However, more protected regions may introduce additional overhead. In our practice, we opted for a *root function* encapsulating all sensitive functions as the protected region, utilizing sMVX to safeguard the root function (e.g., `func2()` in Figure 2).

The instrumented application will be linked against a customized library containing the `mvx_*()` function stubs, while the actual implementation of these functions resides within the sMVX monitor (refer to Section 3.4). During runtime, calls to `mvx_*()` functions are redirected to the sMVX monitor, ensuring that the target application cannot directly access the monitor code. To facilitate this process, we provide a script that analyzes the binary and extracts ELF section, symbol, and function information. It's worth to note that this implementation choice can be entirely handled within the sMVX library.

Running applications with sMVX: Before running the application with sMVX, the end-user will need to run the aforementioned script to create a profile file (in a `/tmp` filesystem). It contains information about the start offsets and size of the `.text`, `.data`, `.bss`, `.plt`, and `.gotplt` sections of the application. We also save the symbol table information to the profile file so the sMVX monitor can resolve the user-specified protection function name (*i.e.,* the

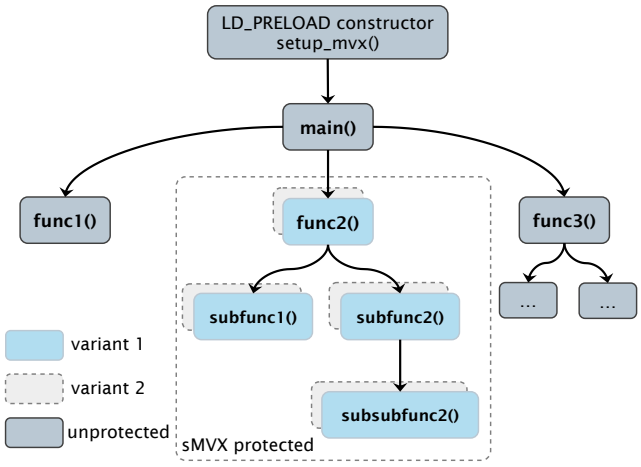

Figure 2. Callgraph of a program protected by sMVX

first parameter of `mvx_start()`) to a function address. This mapping will then be used to ascertain the offset of a selected symbol and instruct the follower variant to begin execution from that point.

When an application needs to be run under sMVX, the end-user loads the sMVX monitor with the `LD_PRELOAD` linker environment variable (*i.e.,* `LD_PRELOAD=./smvx_monitor.so ./<app bin>`). The dynamic loader first runs its constructor function `setup_mvx()` to perform the sMVX setup. Specifically, it stores the original libc symbol addresses so we can use them internally within the library without intercepting ourselves. It also initializes the loader subsystem within the library which loads the follower variant into memory. The `setup_mvx()` also reads the binary information from the previously created `/tmp` filesystem file and retrieves the dynamic process information from `/proc/self/maps`. The constructor also patches the loaded PLT entries of the running process with the MPK trampoline (Section 3.4). The last thing for the constructor is to set up the shared-memory IPC, including the mutexes, condition variables, and libc call emulation-related data.

Figure 2 shows the call graph of an example program protected by sMVX. By placing the `mvx_start()` and `mvx_end()` calls around the call site of the `func2()`, its entire call graph subtree is replicated by sMVX. The `main()`, `func1()` and `func3()` (and by extension its own subnodes) do not have their execution duplicated with a follower variant, only `func2()` and its child nodes run in lock-step with a follower variant. In the rest of this section, we will describe a dynamic analysis method to identify a list of potential sensitive functions (*e.g.,* `func2()` in the example).

Finding an approximate list of sensitive functions with dynamic program analysis: To help the end-user find a minimal yet necessary protected region, we leverage existing dynamic program analysis tools to locate the potential sensitive functions. As mentioned earlier, security sensitive code can be functions that handle external input data or functions that authenticate users' identities. For each case, we provide a semi-automatic method to locate these sensitive functions.

We leverage the dynamic taint analysis to identify functions that handle external inputs. The idea behind it is to track any memory byte tainted by the external input (*e.g.,* through network sockets) and filter out the code that interacts with the "tainted memory". Such tainted memory can be memory buffers that directly store the network input (tainted source) or any memory bytes that the tainted memory flows over (through memory copy, etc.) [42]. We assume the external attacker will likely trigger the exploit remotely by sending payload data to the target process. We leverage `libdft` [23] to locate functions handing external network inputs. `libdft` is a framework to track data flow between the main memory and registers at byte-granularity and provides an API to customize what code and data to instrument [23]. In our extension, we marked the network input as the taint source. As such, network-related memory will be tagged during the program execution and tracked as it is copied and altered by instructions in our target programs. The output of our tool is *a list of instruction addresses that accessed the tainted memory* (originally from the network input). This denotes the code pieces that the remote attacker can externally control.

Next, we wrote a script to get the functions in which those instructions reside. We used the r2pipe [38] in our script to find the functions containing the "tainted instructions" and dump out the symbol names of the function candidates to be protected by sMVX. All steps mentioned above are illustrated in Figure 3. For the Nginx web server, here are some of such functions which handle tainted network data: `ngx_http_handler()`, `ngx_http_header_filter()`, and `ngx_http_process_request_line()`. These will be candidates to be protected with sMVX.

Run libdft on target application

```
pin -follow_execv -t \
libdft/tools/taint_tacking.so -s 0 -f 0 -n 1 -- \
<Application Bin> <params> ...
```

dft.out

dft_result_parser:

```
void main() {
  /* Parse dft.out and filter by .text addresses */
  parse_libdft_output();

  /* Parse target binary using r2pipe
     and get nearest func symbols*/
  parse_target_binary();

  /* Output function symbols to file as candidates */
  dump_function_names();
}
```

Figure 3. Taint analysis workflow

The authentication-related code discovery is a little different from locating tainted data handling code. We leverage the code coverage information collected from a successful user authentication input and a failed authentication input to locate any authentication-related functions. Specifically, we collected two execution trace logs and used the *diff* of two log files as the hint for finding the authentication code. Although multiple basic blocks can be executed differently in the success and failed execution logs, we found that the first divergent basic block is likely to be authentication-related, and functions containing these basic blocks are likely used for authentication. Through these dynamic program analysis methods, we successfully identify the sensitive code.

Emulation requirement	Libc function names
Libc calls only requiring return value emulation	`open, close, shutdown, write, writev, epoll_ctl, setsockopt`
Libc calls requiring return value and argument buffer emulation	`sendfile, stat, read, fstat, gettimeofday, accept4, recv, getsockopt, localtime_r`
Libc calls requiring special emulation	`ioctl, epoll_wait, epoll_pwait`

Table 1. Libc calls emulation with different requirements

3.3 Libc-calls Synchronization and Checkpoint

After the protected region is defined and replicated, sMVX also needs to synchronize the variants' execution, especially system calls (or library calls) that interact with the stateful data. For example, we cannot allow a `write()` system call to be executed by both variants as it will write duplicated data to the same file. The sMVX monitor prevents this duplication by synchronizing data between variants at the libc call boundary. Similarly, the sMVX monitor also needs to synchronize the `read()/send()/recv()` calls and their variants that perform I/O operations. This is because sharing the same filesystem results in a situation where one variant may read and clear a buffer or handle and clear an event (*i.e.,* in the case of `epoll_wait()`), resulting in the other variant arriving at that libc call no longer seeing that data and subsequently causing an execution divergence. There are other cases that will cause false positives if not properly handled. As mentioned in the Orchestra paper [39], such libc calls include time-related calls (*i.e.,* `localtime_r()` and `gettimeofday()`), and data read from `/dev/urandom` and equivalents. sMVX similarly emulates these libc calls.

Besides synchronizing I/O data, the sMVX monitor also checks several parameters for consistency across variants, such as function names that are being called, return values from each libc call, and non-pointer argument values. This ensures the sMVX monitor detects any divergent execution immediately when variants go into different execution paths. Table 1 shows a summary of libc calls with these emulation requirements we supported to run sMVX on the applications showcased in the evaluation (Section 4). All three categories of libc calls needing emulation on the follower variant also require `errno` emulation to correctly give the follower variant accurate information regarding the error status of the libc call.

The first category of libc calls requiring emulation *only for function return values* on top of errno. These libc functions do not write to any application buffers passed in through arguments. The second category of libc calls *write data to the memory buffer pointed by the corresponding arguments*. Therefore, the data can be passed back to the application. When the two variants both call such libc calls, there will be logical memory write issues. To solve that, sMVX adopts a common solution from the past work [9, 25]. Specifically, sMVX only allows the leader variant to execute the libc call. Then, the leader copies over this returning data to the follower variant through the IPC, allowing the follower to skip execution of that libc call but with the returning data passed back to the application.

Finally, there are a few libc calls which require special handling. For example, `ioctl()` has variable arguments depending on what request and device driver it is sending commands to. This results in sMVX being unable to detect if any of these arguments are pointer arguments and if emulation is required. In our approach, we noticed that the only call made to `ioctl()` by our test applications was made in the form where the third argument was a pointer that needed to be emulated and thus only handled that case. One way to handle this in a more robust and generalizable manner would be only to simulate the input argument if it is a pointer that falls within the process's address space. `epoll_wait()` and `epoll_pwait()` are slightly different because whether or not they need to be emulated depends on the mode of `epoll_data` being used. `epoll_data` is a union that can be a file descriptor, 32-bit value, 64-bit value, or a void pointer to data defined by the application. The difficulty arises when the union represents a pointer value as this will need to be emulated on the follower variant. We solved this by checking if the value falls within the process' address space, as with `ioctl()`.

3.4 sMVX monitor

We designed and implemented an sMVX monitor to support above mentioned libc-calls synchronization. The sMVX monitor is implemented as a shared library that resides within the target process space. This requires the monitor code and data to be safely isolated from the target program's code. To this aim, we leverage the Intel MPK hardware feature to ensure the monitor code from being accessed safely (Figure 4). However, a protection key can only be configured to prevent data fetching and cannot prevent code from being executed. Since the monitor code should be isolated from the target program, we adopt a similar approach to MonGuard [54] and use code randomization for hiding the sMVX monitor code. MonGuard is an open-source in-process monitor that leverages the Intel MPK for the monitor isolation. The key idea behind MonGuard is to dynamically update the accessibility of the application memory pages and the monitor memory when control flows transfer between them. In MonGuard, the control flow transfers through a call gate (*i.e.,* a trampoline code). MonGuard uses the execute-only attribute of the memory protection key to prevent the monitor code from being probed [54]. The code monitors are implemented as shared libraries in MonGuard and are loaded at random locations (using `LD_PRELOAD`). As such, monitor calls that go through the procedure linkage table (PLT) are under the monitor's interception.

sMVX further extends MonGuard by adding the multi-threading support and the stack pivoting. As shown in Figure 4, sMVX switches the stack on entering/leaving the trampoline code; thus, it prevents malicious application code from compromising the monitor's stack [13]. Stack pivoting enables an intercepted libc call to have more than six arguments. The x86_64 function call convention [19] dictates that integer arguments `1-6` of a function call are passed through registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` and `%r9`. Any additional arguments are pushed onto the stack. Further, for functions with variadic arguments, the argument number is also stored in `%rax` and call-time. These rules pose a problem for existing MonGuard trampoline code as the libc or monitor calls do not consider the complicated code interception (the gray parts in Figure 4 are from MonGuard, and the colored parts are introduced by sMVX.). With sMVX, whenever the application issues a libc call, it calls the corresponding PLT entry. The sMVX monitor patches the PLT entries so that every libc call will go through the trampoline code. The PLT and trampoline code are marked as execute-only.

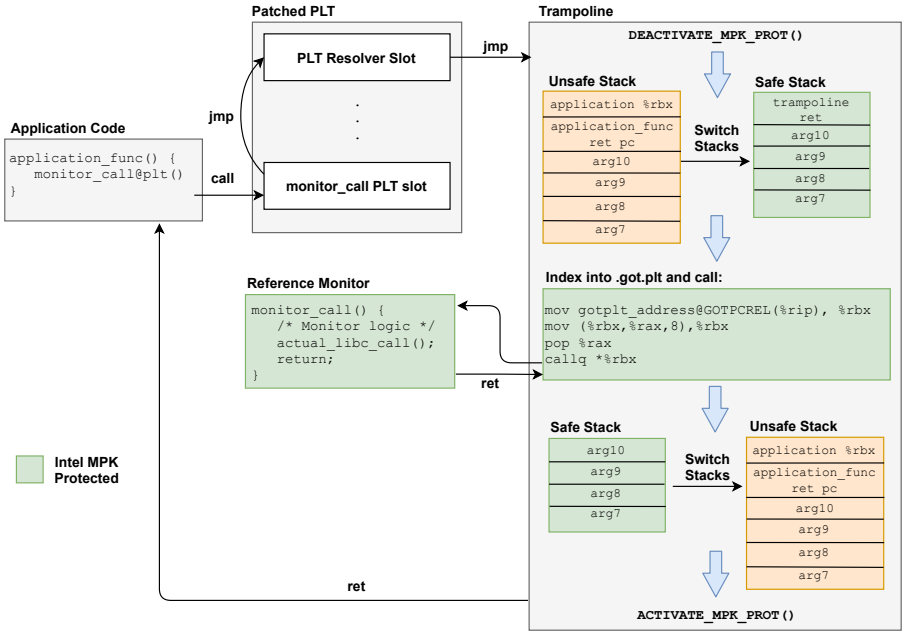

Figure 4. Execution flow of a monitor call (or a library call interception) through the sMVX monitor.

Therefore, an attacker cannot read the PLT code directly to locate the trampoline and the monitor [54]. The sMVX monitor also marks itself and the trampoline data as non-accessible in the application context. On entering the trampoline code, sMVX first enables the data access, switches the call stack, and uses the index of the PLT call to find the actual libc function.

As seen in Figure 4, a newly added `callq *%rbx` instruction redirects the control to the reference monitor resulting in the return address of the trampoline being pushed onto the stack. This also causes any `%rsp` relative addressing within the function code to no longer be semantically correct. Replacing the `callq` instruction with a `jmpq` instruction is not an option as we need the reference monitor function to return to the trampoline so it can reactivate MPK protections and swap back to the unsafe stack. Critically, `%rbx` also needs to be saved by the callee (*i.e.,* the trampoline), limiting the number of registers we are free to use in the MonGuard trampoline to perform the stack pivot and MPK PKRU bit set/reset. To resolve this issue, we rebuild the secure stack in an MPK secured location, replacing the return address to the caller in the application with a return pointer to the trampoline, thereby making the interception completely transparent to the reference monitor and the subsequent libc call. `%rbx` is also saved onto the unsafe stack and will be popped upon returning to the trampoline. The `%rax` value on entering the trampoline is restored and passed into the call to the reference monitor so variadic function calls into the reference monitor remain unbroken (Figure 4). In summary, our extension allows variadic libc calls and libc calls with more than six parameters to be intercepted and emulated.

In order to support multi-threaded applications, the sMVX trampoline's stack memory area and unsafe stack pointer are created as thread local storage (TLS) variables within the sMVX library's address space. This allows each newly-created thread to have its own sMVX trampoline stack when entering the monitor. It's worth noting that the TLS variables utilized by applications are also duplicated during variant creation. Another benefit of using the safe stack is to prevent the untrusted application code from attacking the monitor stack. This is used by several existing attacks where attackers reuse the call stack history and the data left on the stack to generate the exploit [13, 18].

Follower variant creation and pointer relocation: sMVX leverages the non-overlapping address space to create the follower variant. Therefore, the sMVX monitor is also responsible for generating the follower's non-overlapping address space. On entering the protected region (*i.e.,* executing `mvx_start()`), the sMVX monitor creates a thread for the follower variant using the `clone()` system call. On most operating systems, threads share an almost identical address space, except each thread has its own stack region. sMVX thus has to relocate the follower's memory to construct the non-overlapping address space. Moving the code and data in an address space at runtime comes with challenges. These challenges are similar to what is faced by the runtime address space layout re-randomization systems [31, 56] but to a lesser extent. Figure 5 demonstrates this issue of relocating the process address space.

By linking the program as a position independent executable (PIE), the code within the executable's address space accesses its own global data (*i.e.,* `.data` and `.bss`) by using IP-relative addressing. This allows for relocation of the entire process address space, which is what address space layout randomization (ASLR) [11] relies on. This case is depicted with the blue arrow in Figure 5 where instructions in the `.text` section is still able to legally access its `.data`. However, since we perform sMVX during the program execution, there are pointers in global data and the heap regions containing references to the now-unmapped original memory locations. These are shown as red arrows in the figure. This problem can

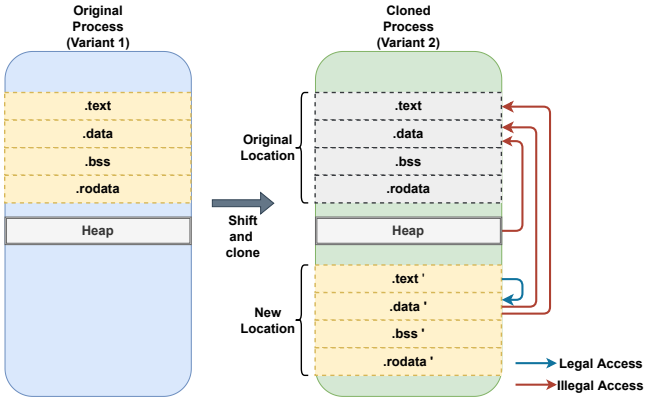

Figure 5. Pointer relocation for the follower variant

be partly solved using pointer tracking [31], compiler-assisted indirection table for code pointers [56] or using the debug information and a customized memory allocator [4].

In sMVX design, we combine the static pointer analysis and runtime pointer scanning. As a result, we use the pointer analysis (*i.e.,* alias analysis) to narrow down the pointer locations and scan the `.data` `.bss` and heap for function pointers to the old `.text` location so execution doesn't attempt to jump back to the old process location. We also need to scan for data pointers pointing at the old `.data` and `.bss` locations to prevent the follower variant from attempting to access illegal memory. Scanning the memory byte-by-byte can be expensive. Fortunately, pointers are 8-bytes aligned on x86_64 systems [34]. In sMVX implementation, we scan each 8-bytes aligned memory slot and use the existing code/data addresses to verify the memory slots are pointers. This is similarly used in RuntimeASLR [31] for false positive pointer identification. We have to note that pointer scanning cannot reliably identify pointers in memory. There might be integer values that look like pointers. sMVX uses this strawman approach to demonstrate the feasibility of selective MVX. Once we have the pointers' addresses, we update these pointers with the appropriate offset to ensure they point to the shifted address space. Note that dynamic memory allocations (*e.g.,* `malloc()`) are allowed on the follower variant after its creation. This is because once the variant is alive, it executes mostly independently from the original program. The follower variant can directly access its newly allocated memory blocks without the need to update memory addresses.

Application domains for sMVX: sMVX is better suited for applications with a distinct division between potentially vulnerable code, such as handling external inputs, and less vulnerable code, like a background program logic. Conversely, programs that intertwine external input handling with core program logic are more suitable for a traditional MVX system.

4 Evaluation

We evaluate the performance impact (including CPU and memory usage) and the security benefit of the sMVX system running on top of the in-process monitor. All of our evaluation was performed on a server with an Intel Xeon(R) Silver 4110 CPU @ 2.10GHz and 92GB of RAM. The server runs Debian 9 with Linux kernel version 5.2.10. sMVX is compiled as a shared library and pre-loaded when running the target application. Currently, the sMVX monitor simulates 35 libc library calls for the follower variant execution.

4.1 Performance Evaluation

We first chose to evaluate sMVX's performance on the Linux/Unix release of the BYTEmark benchmark suite, also known as Nbench [5]. The benchmark includes applications such as Neural Network computation and IDEA encryption algorithm and runs in a single thread. This benchmark shows how sMVX performs when run against applications heavily utilizing the system's CPU, FPU, and memory system. We enclosed the main logic of the benchmark with `mvx_start()/mvx_end()` and conducted three separate runs of the benchmark applications both with and without sMVX. Subsequently, we recorded the mean values of the execution time for each case. Figure 6 reports the overhead of running the benchmark under sMVX. Executing sMVX on CPU-intensive programs gives us very promising performance numbers. As shown in Figure 6, sMVX brings an average of 7% of performance overhead. Applications such as Number Sort, Bitfield, and Assignment perform almost close to the native execution. This is because they are primarily CPU-intensive. Thus, the overhead of the lock-step check is low. The highest overhead seen is the Neural Network benchmark, with about 16% performance slowdown. This is likely due to its relatively high I/O usage (of reading the model file) compared to other tests.

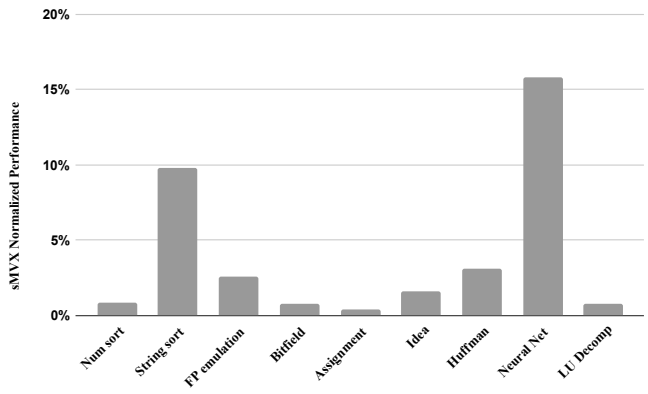

Figure 6. Overhead of running nbench under sMVX.

To evaluate how sMVX performs on I/O heavy and real-world workloads, we measured the performance overhead of sMVX using two server applications. Specifically, we employed the ApacheBench (`ab`) server benchmarking tool as our workload generator, conducting tests on the loopback network interface with a latency of 0.1ms, consistent with that mentioned in a state-of-the-art MVX system – ReMon [8, 49]. We chose to compare sMVX with ReMon because ReMon is widely recognized for its superior performance compared to other MVX systems such as VARAN [17], Orchestra [39], and GHUMVEE [8]. The page size that we were serving from the web servers was 4KB in length. We used the HTTP throughput of vanilla application execution as the baseline to normalize the performance overhead of ReMon and sMVX (Figure 7). With sMVX, we achieve a *266%* overhead for Nginx and a *223%* overhead for Lighttpd.

To understand the reason behind the performance differences, we also measured the total number of libc calls and system calls

issued from each application. We reported the ratio of the number of libc calls versus system calls in Figure 7. For Nginx, there will be about 5.4 libc calls issues over one system call, while that ratio rises to 7.8 for Lighttpd (the y-axis on the right side of Figure 7). The ratio indicates more libc calls are issued than system calls during the program execution[2]. Since ReMon intercepts and emulates systems calls, it suffers less overhead from the emulation (less frequent monitor code invocation). On the other hand, sMVX has to invoke the monitor code more frequently. This is especially the case for Lighttpd, the ratio is up to 7.8, leading to a larger overhead. Furthermore, sMVX currently uses a costly pointer scanning approach to relocate pointers (Table 2). These sources of overhead overshadow the performance gain from selective multi-variant execution. Moreover, ReMon is highly customized for variants' synchronization. Therefore, sMVX cannot ultimately outperform ReMon in terms of performance. However, we envision a different variant creation strategy that can be used to avoid pointer updates. For example, we can create two program variants with varying options of the compiler (*e.g.,* one with forward control-flow integrity enabled and one with shadow stack enabled). This way, we can align the function addresses but still have different variant layouts. We leave performance optimization as future work.

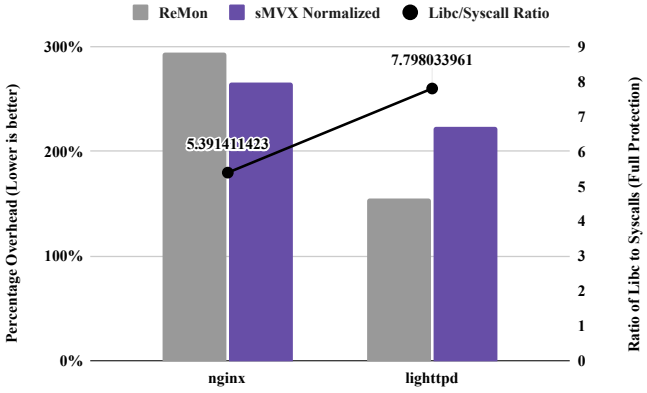

Figure 7. Nginx and Lighttpd performance under sMVX

CPU cycles saved from selective MVX. We also measured the CPU cycles saved by sMVX. Since the follower program variant can be created and destroyed repetitively, it is hard to obtain the overall CPU cycles used by all short-living variants during the runtime. Our experiment calculates the CPU cycles saved by profiling a single variant of the Nginx web server. Specifically, we leverage the Linux profiler tool – perf [24] to profile the CPU cycles used by each function. We further use the flame graph [14] to visualize the call graph and the corresponding CPU usage of each function. We found that the outermost tainted function we identified (*i.e.,* `ngx_http_process_request_line()`) consumes 60.8% total CPU cycles for the `ApacheBench` workload. Other tainted functions identified are within the subtree of that outermost function. Therefore, we can apply sMVX on one function for the best trade-off between the performance and security of the MVX system. Considering we have two variants running under sMVX and a traditional MVX system, the CPU consumption of replicating the Nginx web server is ≈160% versus 200%, correspondingly. We similarly profiled the CPU usage of Lighttpd. We choose the outermost function `server_main_loop()` that contains all the sensitive functions identified. We find that `server_main_loop()` consumes 70% of the total CPU cycles from the flame graph. Therefore, the CPU consumption of replicating Lighttpd with sMVX is ≈170% versus 200% under traditional MVX systems.

Memory consumption saved from selective MVX. We measured the memory consumption of Nginx and Lighttpd both with and without sMVX using the Linux `pmap` command, and recorded the Resident Set Size (RSS) values [55]. The RSS indicates how much RAM a process has been allocated during its execution. Additionally, we replicated the vanilla applications to simulate the memory usage of a traditional MVX system. All measurements were taken after 10 HTTP requests. Running Nginx under sMVX consumes 3208KB of RAM, whereas running two copies of vanilla Nginx consumes 6392KB of RAM[3]. Similarly, running Lighttpd under sMVX consumes 1372KB of RAM, while running two copies of vanilla Lighttpd instances consumes 2720KB (1360KB × 2) of RAM.

The savings in CPU and memory resources stem from the reduced code regions requiring replication. Consequently, this aspect is independent of the specific MVX implementation, whether it be an in-process monitor or an out-of-process monitor. However, when compared to MVX systems that run variants on distinct OS kernels and different machines [9, 25, 53, 59], sMVX notably conserves significantly more CPU and memory resources.

Numbers of libc calls for simulation under different protected functions. To further understand the internal relationship of function calls and MVX-related libc calls are in each selected function call, we measured the number of libc calls when dynamically choosing different functions as the protected region. We started by protecting the entire program call graph (*i.e.,* `main()`), and then shrunk the protected code region and measured the library call numbers in other primary Nginx functions. We collected the number of procedure calls to the PLT (primarily libc library) in Nginx under the benchmark workload with 100k HTTP requests. As we reduce the protected call graph size, fewer libc calls are issued (Figure 8). It means the sMVX monitor only has to emulate a smaller number of libc calls for the follower variant. Consequently, fewer CPU resources are consumed. We expect the performance of sMVX to generally increase, except for functions directly in the control loop of the program, as that would repetitively incur the overhead from process duplication and pointer updates. Note that the purple triangles denote the functions deemed tainted through our taint analysis performed in Section 3.2, showing fewer PLT calls that need to be duplicated and checked over protecting the `main()` function.

The cost of variant creation (pointers relocation). sMVX relies on both static analysis and dynamic pointer scanning to identify the pointers requiring updating (refer to Section 3.4). This occurs upon invoking the `mvx_start()` function. A breakdown of the overheads on the Lighttpd web server is shown in Table 2. We observe that the latency from the process duplication itself is trivial compared to the overheads seen when scanning the heap for code or data pointers pointing to the original `.data`, `.bss` or

[2]This is because some libc calls (*e.g.,* `malloc()`) don't have to issue a system call immediately. Instead, they can serve the application using pre-allocated resources.

[3]We configured Nginx to use 1 master process and 1 worker process. Running Nginx under sMVX takes 1708KB + 1500KB of RAM for the master process and the worker process, respectively. Running two copies of Nginx takes 1704KB + 1492KB + 1704KB + 1492KB of RAM.

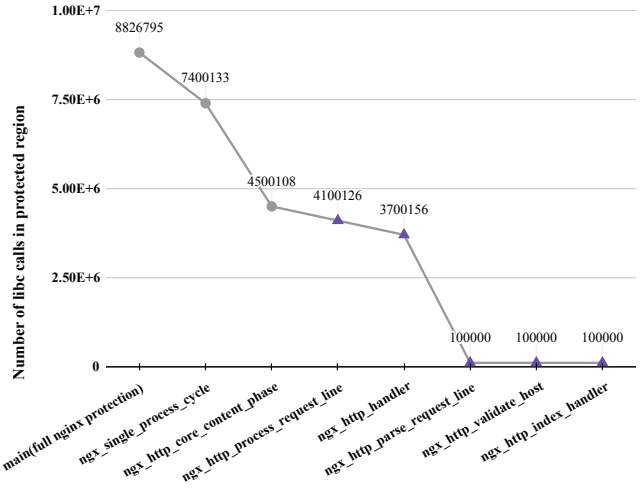

Figure 8. Number of libc calls within protected region (the purple triangles denote the tainted functions).

`.text` locations. Furthermore, the time overhead of the process duplication is akin to vanilla thread creation, approximately around 10 μs (Table 2). In comparison, the process creation with `fork()` imposes an overhead of about 640 μs. When the variant creation is within control loops, we noticed the performance overhead raises high. Similar issue is also found in previous `RuntimeASLR` work [31] and the issue can be similarly solved by pre-scanning and pre-updating the variant. However, when the variant creation is outside of the control loop code path, the overhead comes to normal as we described above.

Source	Latency (μs)
Process duplication (copy+move)	14.7
Data pointer scan overhead	320.8
Heap pointer scan overhead	13162.4
Thread creation with `clone()` (empty function)	9.5
`fork()` overhead (empty `main()` function)	640
`fork()` overhead (during Lighttpd initialization)	697

Table 2. `mvx_start()` overheads on `Lighttpd`

Identify tainted functions through fuzzing. We also evaluated the effectiveness of sensitive functions identified through dynamic taint analysis. We used the standard web server workload generator (`ApacheBench` or `ab`) and a URL fuzzer (`scout`)[4] to generate the workload that covers as many cases as possible. We sent the workload to an Nginx server running on top of the taint analysis engine (described in Section 3.2). We started with the `ApacheBench` workload, and our tool reported 16 sensitive functions, as shown in Figure 9. Next, we ran the `scout` URL fuzzer and checked the number of sensitive functions throughout the whole fuzzing process. `scout` can quickly find a large number of sensitive functions in 5 minutes (Figure 9). It finished the fuzzing task in 41 minutes and generated a 2.92 GB trace file. By parsing the trace log, we found 30 sensitive functions. We also observed that running these workloads on web server applications does not trigger false positives of pointer relocation, which shows a tiny chance of pointer misidentification.

[4] https://github.com/liamg/scout

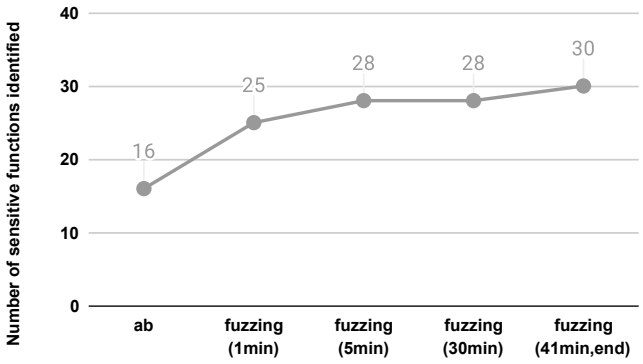

Figure 9. Number of sensitive functions from taint analysis

4.2 sMVX Security Analysis

We take a look at the security guarantees that the sMVX system provides. These security guarantees of sMVX are not that different from those found in other MVX systems [25, 39, 46, 49, 53]. We rely on the variation between two variants to elicit a different response (whether a different libc call sequence or segfault and crashing) with the same (attack) payload. The divergence between Variant 1 and Variant 2 in sMVX is provided by the non-overlapping address spaces of each variant. This results in control-flow-hijacking attacks such as ROP or return-to-libc failing to execute ROP chains containing gadgets found in the program's address space consistently between variants, as the locations of these gadgets will differ, causing memory access errors in Variant 2 when attempting to jump to original gadget locations from Variant 1. Suppose an attacker attempts to modify the program stack data through non-control-data attacks via a buffer overflow vulnerability, causing a divergence in program execution. In that case, the MVX engine will throw a fault and alarm the monitor system due to differing libc function call sequences between the two variants. Such execution divergence can also be detected when comparing the input arguments of libc calls. We now demonstrate sMVX's capability in detecting such exploits using a real-world example:

Nginx CVE 2013-2028: To evaluate the security of sMVX we sought to reproduce a memory corruption vulnerability on one of our guarded applications. In particular, we chose to reproduce a stack-based buffer overflow vulnerability occurring on Nginx, allowing us to perform an out-of-bounds write, overwriting control data of the application, and gain control of the control flow of the web server. The experiment was performed on the Nginx web server version 1.3.9, which contained this vulnerability. The bug in Nginx allowed a remote attacker to specify the size of a chunked HTTP request when issuing a HTTP request with a header "Transfer-Encoding:chunked". This value is an unsigned number, which can later be misrepresented as a negative integer when cast to a signed type. The integer can later be re-casted from a negative signed value to an unsigned integer and used as the data size to read a buffer. Specifically, the target function being exploited, `ngx_http_read_discarded_request_body()`, contains a buffer that will receive data from the attacker via the `recv()`

libc library call, with the goal being to trick `recv()` to perform a buffer out-of-bounds write into that buffer. Therefore, the variable `r->headers_in.content_length_n` is indirectly under attacker's control, ultimately allowing it to be a negative number which when casted to `size_t`, becomes a large positive number. More details can be found in [52]. This buffer overflow allows us to begin a return-oriented programming attack on Nginx. In our exploit, we performed a simple ROP chain using gadgets found in the program address space utilizing the Ropper [41] and ROPGadget [22] tools. Since we are not attempting to prove the turing-completeness of ROP chains, the actual logic being performed is of less importance than its observability. Our ROP chain consists of 3 gadgets and 3 values, loading a pointer to a string found in the application to `%rdi`, popping an integer from the stack to `%rsi`, and jumping to the location of the `mkdir` libc call to create a directory.

Running the exploit on Nginx protected by sMVX, we observe that the follower variant throws a fault when the program counter tries to jump to gadget locations that were present in the leader variant's address space but were otherwise unmapped in the follower variant. Thereby, sMVX detects and breaks the attack. In addition, we should note that `recv()` is one of the I/O libc calls simulated on the follower variant. This allows the in-process monitor to perform extra bounds checks on sensitive calls in the future to prevent such attacks from occurring. We also examined other Nginx CVEs, such as CVE-2016-4450 and CVE-2017-7529. Since these CVEs appear in different Nginx versions and fewer existing exploits are publicly available, we did not reproduce them on the sMVX-protected Nginx instance. Instead, we manually examined them and confirmed the vulnerable functions are on the sensitive code paths and should be detected by sMVX due to the non-overlapping address space.

5 Limitations and Future Works

sMVX uses `LD_PRELOAD` to launch the in-process sMVX monitor to the process's address space. Therefore, it does not support statically-linked applications or applications with the `setuid`/`setgid` permission. sMVX provides an API for application programmers to annotate the protected code region. A dynamic taint analysis method is also proposed to guide the code annotation. However, the dynamic taint analysis results heavily depend on the selected test input and it is difficult to obtain an accurate sensitive region. As a result, the actual attack payload may touch functions beyond the protected code region (a false negative in exploit detection). One solution is to leverage state-of-the-art test case generation techniques, such as fuzzing [44] to generate as many test cases as possible. We anticipate that combining fuzzing techniques with the taint analysis might help increase the code coverage and obtain a more accurate set of sensitive functions. Another possible solution is to combine the dynamic taint analysis result and the static program analysis to find a more accurate set of sensitive functions [12, 35]. Static value flow (SVF) [43] analyzes the load and store instructions in the middle-end compiler representation (*i.e.,* LLVM IR) to track sensitive value flows. Specifically, it can statically analyze the memory value locations derived from the tainted input and infer the sensitive functions. However, the static analysis may calculate a larger (overestimate) sensitive function set. Therefore, obtaining an accurate list of sensitive functions is our future work.

When running sMVX and protecting functions within event loops, we ran into a performance issue - the overheads involved in the setup of the second variant dominate the savings we receive from not having to perform MVX lockstep execution on untainted portions of the application. Although we used the efficient `clone()` system call with memory movements to generate the follower variant, the event loop amplifies this cost. We observe that the latency from duplicating the process itself is trivial compared to the overheads seen when scanning the heap for code or data pointers pointing to the original `.data`, `.bss` or `.text` locations. This pointer updating issue was also mentioned in literature such as RuntimeASLR [31]. Because of this, we anticipate sMVX may not be suitable for pointer-intensive applications, such as graph processing and data searching applications, especially for massive pointers processing within sensitive regions. However, sMVX may be more practical to systems with a clear separation of front-end (receiving and processing user input) and back-end code logic. Merely replicating the front-end code can achieve better security at a lower performance penalty, fewer CPU cycles, and less memory usage than directly applying MVX techniques to the whole system.

One possible way to efficiently track the pointers at runtime is to leverage the upcoming hardware tagged memory [2, 21]. Specifically, we can associate different tag values to distinguish pointers from normal data and leverage the hardware to track pointer value flows. Besides leveraging the upcoming hardware, another approach is to generate an indirect jump table for all pointers [3, 56]. The indirection table contains the actual pointer target addresses. One challenging problem of this approach is to identify data pointers allocated from dynamically allocated memory (*e.g.,* from `malloc()`). One solution is to statically analyze memory allocations that contain data pointers and replace the corresponding `malloc()` with a customized one. The customized `malloc()` replaces the data pointer to an address of an indirection table. Therefore, the sMVX monitor only needs to update each entry in the indirection table instantly instead of finding all pointers through the whole memory. There may be other ways to create a diversified follower variant more efficiently. For example, we may only shuffle the order of basic blocks within the target functions but keep function addresses unchanged to avoid updating code pointers. We leave these potential optimizations as future works.

6 Related Work

The first category of related work is the *various systems on multi-variant execution.* In general, the MVX monitor plays an important role in ensuring the correct and safe execution of the variants. There are multiple design choices for the MVX monitor [49]. Some MVX systems place the system-call interception monitors outside the target process for more robust isolation through, for example, the `ptrace` interface [16, 39, 50]. The cross-process system call monitoring enables strong memory isolation between the MVX monitor code and the target. However, they may suffer from large performance overhead. On the contrary, some MVX monitors within the target process's address space often perform better [17, 54]. One such monitor is VARAN [17], which locates in userspace as a statically linked library but performs system call interception by rewriting the binary and patching system calls in the application binary to redirect the execution to dedicated handlers. ReMon [49] combines an (insecure) in-process monitor and a (secure) cross-process

monitor to achieve the best trade-off of security and near-native performance. Although it reduces the performance overhead of the MVX system, ReMon (and existing MVX systems) still runs multiple program variants entirely and thus consumes multiple times of CPU and memory usage [49].

Other MVX monitors reside within the operating system kernel, so they are isolated from the program variant and require fewer context switches [9, 25, 53, 59]. N-Variant [9] is an in-kernel monitor and is the pioneering paper in this field. It uses a heterogeneous memory space and instruction set tagging where the instructions are pre-pended with a tag bit to create diversity between variants. HeterSec [53] is an MVX system working across heterogeneous Instruction Set Architectures (ISA). The variance in ISA and stack layout between architectures guarantees that an attack payload working on a particular variant does not replicate its effects on another variant on another architecture. kMVX [59] also uses an in-kernel monitor but focuses on protecting the kernel from information leak vulnerabilities. In contrast, we do not aim to provide a new MVX technique or a different strategy for variant creation; instead, we aim to reduce the resource consumption of MVX systems by selectively replicating sensitive code regions.

The second category of related work is the recent effort in *intra-process isolation and dynamic protection*. Researchers have proposed using operating system primitives [29], hardware primitives [26, 27, 58] and even virtualization techniques [30] to protect sensitive data inside the address space. For example, light-weight contexts (lwCs) modifies the OS kernel to provide independent units of isolation within a process [29]. LOTRx86 repurposed the unused privilege rings (*i.e.,* ring 1 and ring 2) to achieve the intra-process privilege separation [26]. ARMlock leverages the ARM memory domain to isolate untrusted library code from being maliciously used [58]. These systems achieve intra-process isolation by providing various memory views to different process components. Following this direction, recent work leverage the Intel MPK to achieve sensitive data isolation with cheaper performance cost [15, 20, 36, 45, 54]. For example, libmpk is proposed as a library to protect sensitive data and virtualize the protection keys to solve the scalability problem [36]. ERIM utilizes binary inspection and rewriting to prevent unintended sensitive MPK instructions from being maliciously used [45]. sMVX extends this direction by implementing a secure in-process MVX monitor with a per-thread, MPK-protected safe stack. sMVX also adopts the concept of selective software protection [33, 35]. Particularly, DynPTA combines static and dynamic analysis to selectively protect sensitive data [35]. DynaCut dynamically eliminates unused code features across various software execution phases to minimize the attack surface [33]. sMVX aligns with this research line but focuses on reducing the CPU and memory overhead of MVX systems.

7 Conclusion

We have presented sMVX, a system for multi-variant execution on selected code paths. sMVX allows the end-user to define a protection region to be replicated. Users can apply only three lines of code instrumentation to indicate the sMVX monitor for creating the variant of selected sensitive code paths. Moreover, sMVX also provides a dynamic taint analysis tool to identify security-sensitive functions in an application semi-automatically. A prototype of sMVX was built with a hardware-secured in-process code monitor. The evaluation shows sMVX can achieve similar performance overhead as the state-of-the-art MVX monitor but require 20% fewer CPU cycles and 49% less memory consumption.

Acknowledgments

We thank Dr. R. Sekar, the anonymous reviewers, and our shepherd, Dr. Marios Kogias, for their insightful comments, which have greatly improved the paper. This work is partly supported by the US Office of Naval Research under grants N00014-18-1-2022, N00014-19-1-2493, and N00014-22-1-2672, and by the US National Science Foundation (NSF) under grant CNS 2127491. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

References

[1] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. 2014. You Can Run but You Can'T Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*.

[2] Steve Bannister. 2019. Memory Tagging Extension: Enhancing memory safety through architecture. Retrieved 06/23/2022 from https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety

[3] Sandeep Bhatkar, Daniel C DuVarney, and R Sekar. 2005. Efficient Techniques for Comprehensive Protection from Memory Error Exploits.. In *USENIX Security Symposium*, Vol. 10. 1251398–1251415.

[4] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 268–279.

[5] BYTEmark benchmark. Accessed: 2024-03-20. Linux/Unix nbench. http://www.math.utah.edu/~mayer/linux/bmark.html.

[6] Mengchen Cao, Xiantong Hou, Tao Wang, Hunter Qu, Yajin Zhou, Xiaolong Bai, and Fuwei Wang. 2019. Different is Good: Detecting the Use of Uninitialized Variables through Differential Replay. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 1883–1897. https://doi.org/10.1145/3319535.3345654

[7] Liming Chen and A. Avizienis. 1995. N-VERSION PROGRAMMINC: A FAULT-TOLERANCE APPROACH TO RELIABILITY OF SOFTWARE OPERATION. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'*. 113–. https://doi.org/10.1109/FTCSH.1995.532621

[8] Bart Coppens, Bjorn De Sutter, and Stijn Volckaert. 2018. *Multi-variant execution environments*. Association for Computing Machinery and Morgan & Claypool, 211–258. https://doi.org/10.1145/3129743.3129752

[9] Benjamin Cox and David Evans. 2006. N-Variant Systems: A Secretless Framework for Security through Diversity. In *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006*, Angelos D. Keromytis (Ed.). USENIX Association.

[10] David Mulnix. Accessed: 2024-03-23. Intel® Xeon® Processor Scalable Family Technical Overview. https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview.

[11] Jake Edge. 2013. Linux Kernel Address Space Layout Randomization. http://lwn.net/Articles/569635/.

[12] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 1749–1766.

[13] Enes Göktas, Benjamin Kollenda, Philipp Koppe, Erik Bosman, Georgios Portokalidis, Thorsten Holz, Herbert Bos, and Cristiano Giuffrida. 2018. Position-independent code reuse: On the effectiveness of aslr in the absence of information disclosure. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 227–242.

[14] Brendan Gregg. 2021. CPU Flame Graphs. Retrieved 06/23/2022 from https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html

[15] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 489–504.

[16] Petr Hosek and Cristian Cadar. 2013. Safe software updates via multi-version execution. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 612–621.

[17] Petr Hosek and Cristian Cadar. 2015. VARAN the Unbelievable: An Efficient N-version Execution Framework. In *Proceedings of the Twentieth International*

Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'15, Istanbul, Turkey, March 14-18, 2015, Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas (Eds.). ACM, 339–353.

[18] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 969–986.

[19] Intel 2019. *Intel 64 and IA-32 Architectures Software Developerś Manual.* Intel.

[20] Mohannad Ismail, Jinwoo Yom, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. 2021. VIP: Safeguard Value Invariant Property for Thwarting Critical Memory Corruption Attacks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS'21)*. Association for Computing Machinery.

[21] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N. M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey D. Son, and A. Theodore Markettos. 2017. Efficient Tagged Memory. In *2017 IEEE International Conference on Computer Design, ICCD 2017, Boston, MA, USA, November 5-8, 2017*. IEEE Computer Society, 641–648. https://doi.org/10.1109/ICCD.2017.112

[22] JonathanSalwan. 2020. Ropgadget Github webpage. https://github.com/JonathanSalwan/ROPgadget, Online accessed 2024-03-23.

[23] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. 2012. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*. 121–132.

[24] kernel.org. 2020. perf: Linux profiling with performance counters. Retrieved 06/23/2022 from https://perf.wiki.kernel.org/index.php/Main_Page

[25] Koen Koning, Herbert Bos, and Cristiano Giuffrida. 2016. Secure and Efficient Multi-Variant Execution using Hardware-Assisted Process Virtualization. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 431–442.

[26] Hojoon Lee, Chihyun Song, and Brent Byunghoon Kang. 2018. Lord of the X86 Rings: A Portable User Mode Privilege Separation Architecture on X86. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1441–1454. https://doi.org/10.1145/3243734.3243748

[27] Congwu Li, Le Guan, Jingqiang Lin, Bo Luo, Quanwei Cai, Jiwu Jing, and Jing Wang. 2019. Mimosa: Protecting Private Keys against Memory Disclosure Attacks using Hardware Transactional Memory. *IEEE Transactions on Dependable and Secure Computing* (2019).

[28] Linux. 2020. pkeys(7) — Linux manual page. https://man7.org/linux/man-pages/man7/pkeys.7.html.

[29] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 49–64.

[30] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 1607–1619.

[31] Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. 2016. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society.

[32] Kangjie Lu, Meng Xu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2021. Stopping Memory Disclosures via Diversification and Replicated Execution. *IEEE Trans. Dependable Secur. Comput.* 18, 1 (2021), 160–173. https://doi.org/10.1109/TDSC.2018.2878234

[33] Abhijit Mahurkar, Xiaoguang Wang, Hang Zhang, and Binoy Ravindran. 2023. DynaCut: A Framework for Dynamic and Adaptive Program Customization. In *Proceedings of the 24th International Middleware Conference*. 275–287.

[34] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. 2013. System v application binary interface. (2013).

[35] Tapti Palit, Jarin Firose Moon, Fabian Monrose, and Michalis Polychronakis. 2021. DynPTA: Combining Static and Dynamic Analysis for Practical Selective Data Protection. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1919–1937.

[36] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 241–254.

[37] Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. 2019. MVEDSUA: Higher Availability Dynamic Software Updates via Multi-Version Execution. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 573–585. https://doi.org/10.1145/3297858.3304063

[38] Radare Org. 2024. R2pipe Github webpage. https://github.com/radareorg/radare2-r2pipe, Online accessed 2024-01-23.

[39] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. 2009. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 33–46.

[40] Salamat, Babak and Gal, Andreas and Franz, Michael. 2008. Reverse stack execution in a multi-variant execution environment. In *Workshop on Compiler and Architectural Techniques for Application Reliability and Security*. 1–7.

[41] Sascha Schirra. 2024. Ropper Github webpage. https://github.com/sashs/Ropper, Online accessed 2024-03-23.

[42] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004*, Shubu Mukherjee and Kathryn S. McKinley (Eds.). ACM, 85–96.

[43] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. ACM, 265–266.

[44] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: brute force vulnerability discovery*. Pearson Education.

[45] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient in-Process Isolation with Protection Keys (MPK). In *Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19)*. USENIX Association, USA, 1221–1238.

[46] Jonas Vinck, Bert Abrath, Bart Coppens, Alexios Voulimeneas, Bjorn De Sutter, and Stijn Volckaert. 2022. Sharing is caring: secure and efficient shared memory support for MVEEs. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis (Eds.). ACM, 99–116.

[47] Stijn Volckaert, Bart Coppens, and Bjorn De Sutter. 2016. Cloning Your Gadgets: Complete ROP Attack Immunity with Multi-Variant Execution. *IEEE Trans. Dependable Secur. Comput.* 13, 4 (2016), 437–450.

[48] Stijn Volckaert, Bart Coppens, Bjorn De Sutter, Koen De Bosschere, Per Larsen, and Michael Franz. 2017. Taming Parallelism in a Multi-Variant Execution Environment. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic (Eds.). ACM, 270–285.

[49] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. 2016. Secure and Efficient Application Monitoring and Replication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 167–179.

[50] Alexios Voulimeneas, Dokyung Song, Per Larsen, Michael Franz, and Stijn Volckaert. 2021. dMVX: Secure and Efficient Multi-Variant Execution in a Distributed Setting. In *Proceedings of the 14th European Workshop on Systems Security*. 41–47.

[51] Alexios Voulimeneas, Dokyung Song, Fabian Parzefall, Yeoul Na, Per Larsen, Michael Franz, and Stijn Volckaert. 2020. Distributed Heterogeneous N-Variant Execution. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24-26, 2020, Proceedings*, Clémentine Maurice, Leyla Bilge, Gianluca Stringhini, and Nuno Neves (Eds.), Vol. 12223. Springer, 217–237.

[52] w00d. 2013. Analysis of nginx 1.3.9/1.4.0 stack buffer overflow and x64 exploitation (CVE-2013-2028)). https://www.vnsecurity.net/research/2013/05/21/analysis-of-nginx-cve-2013-2028.html.

[53] Xiaoguang Wang, SengMing Yeoh, Robert Lyerly, Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. 2020. A Framework for Software Diversification with ISA Heterogeneity. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2020, San Sebastian, Spain, October 14-15, 2020*, Manuel Egele and Leyla Bilge (Eds.). USENIX Association, 427–442.

[54] Xiaoguang Wang, SengMing Yeoh, Pierre Olivier, and Binoy Ravindran. 2020. Secure and efficient in-process monitor (and library) protection with Intel MPK. In *Proceedings of the 13th European Workshop on Systems Security, EuroSec@EuroSys 2020, Heraklion, Greece, April 27, 2020*. ACM, 7–12.

[55] Wikipedia. Accessed: 2024-03-20. Resident set size. https://en.wikipedia.org/wiki/Resident_set_size.

[56] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and Deployable Continuous Code Re-Randomization.. In *OSDI*. 367–382.

[57] Xiantao Zhang, Xiao Zheng, Zhi Wang, Qi Li, Junkang Fu, Yang Zhang, and Yibin Shen. 2019. Fast and Scalable VMM Live Upgrade in Large Cloud Infrastructure. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. ACM, 93–105.

[58] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. 2014. ARMlock: Hardware-Based Fault Isolation for ARM. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 558–569.

[59] Sebastian Österlund, Koen Koning, Pierre Olivier, Antonio Barbalace, Herbert Bos, and Cristiano Giuffrida. 2019. kMVX: Detecting Kernel Information Leaks with Multi-variant Execution. In *ASPLOS*.