Distributed Heterogeneous N-Variant Execution

Alexios Voulimeneas[1](✉), Dokyung Song[1], Fabian Parzefall[1], Yeoul Na[1], Per Larsen[1], Michael Franz[1], and Stijn Volckaert[2]

[1] Department of Computer Science, University of California, Irvine, USA
{avoulime,dokyungs,fparzefa,yeouln,perl,franz}@uci.edu
[2] Department of Computer Science, imec-DistriNet, KU Leuven, Leuven, Belgium
stijn.volckaert@cs.kuleuven.be

Abstract. N-Variant Execution (NVX) systems utilize artificial diversity techniques to enhance software security. The general idea is to run multiple *different* variants of the same program alongside each other while monitoring their diverging behavior on a malicious input. Existing NVX systems execute diversified program variants on a *single* host. This means the level of inter-variant diversity will be limited to what a single platform can offer, without costly emulation. This paper presents DMON, a novel distributed NVX design that executes native program variants across *multiple* heterogeneous hosts. Our approach greatly increases the level of diversity between the simultaneously running variants that can be supported, encompassing different ISAs and ABIs. Our evaluation shows that DMON can provide comparable performance to traditional, non-distributed NVX systems, while enhancing security.

1 Introduction

Memory errors are a continuous source of software vulnerabilities for C and C++ programs. Attackers and defenders are engaged in an arms race in which the latter keep developing sophisticated defenses while their adversaries create new exploits that bypass these defenses [46]. At present, adversaries rely on intimate knowledge of the target environment to mount code-reuse [7,42,43] or data-oriented attacks [9,22,23] that allow them to take control of the target or leak its sensitive data. While memory safety techniques protect against these threats, many of these techniques have not seen widespread deployment due to performance [34,35] and compatibility problems [44]. Instead, defenders resort to mitigations that have a more reasonable performance impact, e.g., control-flow integrity (CFI) techniques [1,6], automated software diversity techniques [28], or a combination thereof. However, both classes of defenses have a history of known weaknesses: CFI defenses often still leave some leeway to mount control-flow hijacking attacks [11,14,48]. Software diversity techniques have been bypassed using brute-forcing and information leakage attacks, including attacks enabled by micro-architectural side channels [4,18,24,43].

C. Maurice et al. (Eds.): DIMVA 2020, LNCS 12223, pp. 217–237, 2020.
https://doi.org/10.1007/978-3-030-52683-2_11

N-Variant eXecution (NVX) systems amplify the effectiveness of software diversity techniques and increase resilience [3,5,12,20,21,25–27,31,32,36,39,50, 52–55]. An NVX system runs multiple diversified variants of the same program in parallel on the same inputs while monitoring the variants' behavior for divergences. With the right selection of diversity techniques, NVX can make exploitation substantially harder (and, in some cases, provably impossible) as it forces adversaries to simultaneously compromise multiple program variants without causing observable changes in their behavior. Existing NVX systems have been particularly effective at stopping attacks that rely on knowledge of the target's absolute virtual address space layout [5,12,52], as well as attacks that attempt to acquire that knowledge through information leakage [31]. However, these systems are not resilient to Position-Independent Return-Oriented Programming (PIROP) attacks [16] and certain Data-Oriented Programming (DOP) attacks [23], which build on knowledge of the program's internal geometry (e.g., relative data/instruction layouts) and/or data representation. The main reason is that in previous NVX systems all the variants run on the same machine. Thus, the amount of diversity that such systems can achieve is limited to what a single platform can offer, without using costly emulation.

In this paper, we present DMON, an NVX system that leverages the diversity that naturally exists across different platforms, thereby increasing resilience to memory exploits. DMON runs each program variant natively on its own dedicated machine and monitors divergent behavior between these distributed variants by cross-checking them at the system call boundary via a network. To bypass DMON, adversaries would need to develop exploits that work simultaneously against the two or more different Instruction Set Architectures (ISAs) and Application Binary Interfaces (ABIs) for which the program variants are compiled.

Our contributions are as follows:

– We present DMON, the first system that combines ISA and ABI heterogeneity with N-Variant Execution. DMON distributes the execution of a set of variants over a heterogeneous set of physical machines.
– We redefine *semantic equivalence* of system calls in the context of heterogeneous platforms. Based on this definition, we propose Platform-Independent State Canonicalization (PISC), a novel technique that translates system call states in different platforms into platform-independent states.
– We present ways to reduce the performance overheads associated with a distributed NVX system. Our results show that with the proposed performance optimizations, DMON performs on par with traditional `ptrace`-based NVX systems while providing stronger security guarantees.
– We evaluate DMON's security on several server applications and show that DMON makes code-reuse attacks substantially more difficult, and that DMON naturally provides a high degree of structure layout diversity which raises the bar for attacks that rely on consistent structure layout.

2 Background

Researchers in the information security [3,5,12,26,31,36,39,50,52–55] and software reliability communities [20,21,25,27,32,38] have presented over a dozen different NVX systems since 2006. Although serving different purposes, they do have some essential similarities. All systems have the same high-level architecture; two or more software variants execute simultaneously on the same physical machine, while a monitoring component (on that same machine) compares the variants' overall behavior, provides them with identical inputs, and demultiplexes their outputs. Most monitors force the variants to execute in lock-step at the granularity of system calls. Thus, the variants are suspended at every system call entry and exit, and do not proceed until the system call numbers and arguments have been cross-checked across all variants. In addition, the majority of existing NVX systems cross-check behavior and replicate I/O by intercepting the variants' system calls. Most early systems used a dedicated monitoring process that attaches to the variants and intercepts their system calls using the `ptrace` API [5,20,32,39,53]. To avoid the high run-time performance overhead incurred by context switching between a variant process and the monitor process, several teams explored alternative designs that use binary rewriting [21], virtualization features [26], or kernel modules [12,31,52,54] to intercept and cross-check system calls more efficiently, within the variants' processes and address spaces.

2.1 System Calls and I/O Replication

Most NVX systems monitor behavior and replicate I/O at the system call interface. This design lets the system monitor all behavior that can affect the integrity of the OS or other processes, as well as all communication between the variants and external entities. The monitoring and replication must be transparent to the program variants and to the end-user, i.e., no observable functional differences between native execution of a single variant and NVX of multiple variants. To provide this guarantee, most NVX systems designate one variant as the leader, while the others become followers. Whenever the variants attempt an I/O operation, the NVX systems ensure that only the leader variant actually completes the operation, while the followers skip the operation and wait until they receive the I/O results from the monitor.

2.2 ISA and ABI Heterogeneity

An underlying assumption of most NVX systems is that the program variants will behave identically at the system call level when they receive equivalent, benign inputs. This assumption no longer holds in our setting, where variants run on processors with different Instruction Set Architectures (ISAs). Differences in the endianness, register and pointer width, and the available system calls could lead to observable (yet benign) differences in the variants' behavior, which would all cause false alarms in a traditional NVX system. In addition, the Application Binary Interface (ABI) documents rules such as sizes of primitive data types, struct packing, calling conventions, etc. Many of these conventions also affect the program behavior as observed from the system call interface. Therefore, we had to carefully design DMON to tolerate divergences arising from the heterogeneous ISA and ABI setting.

3 Threat Model

Throughout the rest of the paper, we will make the following assumptions about the host system and the attacker. Our assumptions are consistent with related work in this area [52].

Host Defenses. We assume that the standard set of mitigations are in place on any of the physical machines DMON and the variants run on. Specifically, we assume that Data Execution Prevention (DEP) is used, which therefore rules out direct code-injection attacks. Likewise, we assume that all of the host systems have Address Space Layout Randomization (ASLR) enabled. ASLR randomizes the base addresses of the main program executable and shared libraries, as well as the heap, stack, and any other mapped memory regions.

Remote Attacker. We assume that the attacker does not have direct physical access to any of the machines DMON (or the variants) run on. The attacker can only communicate with the protected application running on the leader machine via a remote communication channel such as a network socket. The followers are connected to the leader through a secure private network connection. The adversary can, therefore, not communicate with the follower variants. We also consider attacks on this private connection to be outside the scope of this paper. Because the attacker is remote, we also assume that any run-time secrets embedded into the variants (e.g., randomized base addresses) are not known a priori. The goal of the attacker in this scenario is to take control of the leader variant, e.g., by exploiting a memory-corruption vulnerability. We assume that the protected application has an arbitrary memory read/write vulnerability that the attacker knows how to trigger.

4 DMON Design

DMON orchestrates and supervises the execution of a set of diversified program variants running natively on machines that differ in their instruction set architecture. Like most other NVX systems, DMON uses a leader/follower-model for I/O replication. The designated leader variant is the only variant allowed to perform externally observable I/O operations such as sending or receiving data from a network socket. DMON forces follower variants to skip these I/O operations and instead provides them with the leader's I/O results, thus emulating the original operation unbeknownst to the follower.

Similar to other security-focused NVX systems such as ReMon [52] and MvArmor [26], DMON executes all security sensitive system calls in lock step. Whenever the variants attempt to execute a sensitive system call, DMON ensures

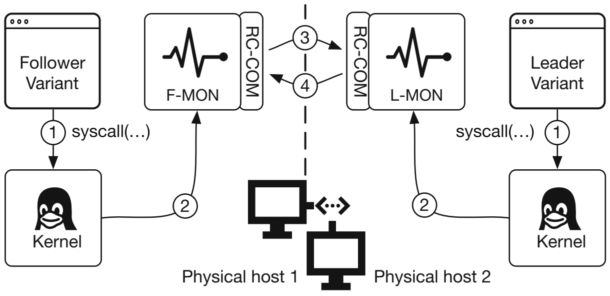


Fig. 1. DMON's basic components and interactions.

that the variants can neither enter the system call routine, nor exit from it until DMON has ensured that all variants have reached equivalent states. We distinguish between the following components of a running DMON system:

1. **Leader Variant.** Only the designated leader variant is allowed to perform externally observable I/O. As in any other NVX system, DMON requires that there is exactly one leader variant.
2. **Follower Variants.** Follower variants skip externally observable I/O operations and use the leader's I/O results instead.
3. **Monitors.** DMON uses two types of monitors: the (single) L-MON monitor supervises the leader variant, while every follower variant is supervised by its own F-MON monitor.
4. **RC-COM.** A reliable communication component used to exchange system call metadata between the monitors. Separating the communication logic into its own layer lets the monitors communicate over a variety of channels.

These components interact whenever the variants execute system calls, as shown in Fig. 1. Whenever a leader or follower variant attempts to enter or exit from a system call (①), the corresponding L-MON or F-MON interrupts and suspends the variant, reads the call number of the interrupted system call, and invokes a specialized handler routine within the monitor process (②), which implements the cross-checking and replication logic for that system call.

The monitors use cross-checking handlers when they interrupt variants upon entering a system call. In F-MON, the cross-checking handler gathers information about the variant's state, sends this information to L-MON (③), and waits for L-MON to confirm that the follower variant is in a state equivalent to the leader variant (④). In L-MON, the cross-checking handler waits for incoming state information from F-MON, compares that state information with the leader variant's state, and informs F-MON about the results of the comparison.

The state information consists of system call numbers and arguments, with the latter often consisting of pointers to complex data structures (e.g., I/O vectors). The cross-checking handlers serialize these corresponding data structures and append the serialized data to the state information, thereby allowing L-MON to check the variant states for deep equivalence (two data structures are deeply equivalent when the raw data they contain is identical, even though the data or the data structures may be stored at different addresses). If the variant states do not match, DMON takes that as a sign of potential compromise and aborts execution to protect the host system.

Naive cross-checking of these variant states triggers false alarms for divergent behavior because the system call interfaces, calling conventions, etc. differ across platforms. DMON transforms system call states to platform-independent states before comparing them, to avoid alarms for the expected platform differences (see Sect. 4.1). If the states match, the cross-checking handler allows the leader variant to proceed and to enter the kernel-space system call routine. The follower variants can also proceed, but may (optionally) see their system call number replaced by that of the `sys_getpid` routine in case they attempt to perform an externally observable I/O operation. This mechanism for skipping system calls was also used in prior work [39]. The monitors use replication handlers when they interrupt variants that return from a system call. Replication handlers for I/O system calls broadcast the system call results from the leader variant to the followers. Replication handlers for other system calls are generally no-ops.

4.1 Platform-Independent State Canonicalization

In traditional NVX systems, all program variants are compiled for the same target architecture and execute on a single machine. In DMON, on the other hand, individual variants run on different physical machines and thus the variants may target different ISAs/ABIs. Heterogeneous platforms expose different system call interfaces. Without awareness of this heterogeneity, cross-checking at this interface leads to false alarms, where the NVX system detects divergence despite the program behavior being equivalent. We find that the root cause of this type of false alarm is the lack of understanding of *semantic equivalence* of system calls in the presence of heterogeneous platforms. To broaden this understanding, we define *semantic equivalence* of system calls as follows:

Definition 1. *The functionality of a syscall is a transformation of one user-observable system state to the other, which constitutes observable behavior. We do not consider behavior observable if it is only visible through side-channels.*

Definition 2. *Given a syscall* $(\boldsymbol{c}, \boldsymbol{p})$*, where* $\boldsymbol{c}$ *is a vector of configuration parameters and* $\boldsymbol{p}$ *is a vector of data parameters, a unique* $\boldsymbol{c}$ *on platform* A *determines the functionality of the system call, which we denote as* $F(\boldsymbol{p})$*.* $\boldsymbol{c}$ *includes the system call number, as well as any flags, modes, etc. that the syscall accepts as parameters to configure its behavior.*

Definition 3. $F(\boldsymbol{p})$ *and* $F'(\boldsymbol{p'})$ *are semantically equivalent iff* F *and* F' *are mapped to the same system call functionality and parameters* $\boldsymbol{p}$ *and* $\boldsymbol{p'}$ *are identical in a serialized form.*

Based on the definition of semantic equivalence, we introduce a technique called platform-independent state canonicalization (PISC), which marshalls syscall states into a canonical syscall state. To do so, DMON internally maintains a canonical representation of system call functionalities and serialization rules. By cross-checking this canonical state, DMON eliminates false positive detections that stem from ABI/ISA heterogeneity.

Semantically equivalent system calls must be mapped to the same canonical system call state. To preserve this property, we define a set of rules that DMON should follow to perform platform-independent state canonicalization (PISC).

Rule 1 - Configuration Constant Canonicalization. According to our definition of semantic equivalence, the configuration parameters of a system call ($\boldsymbol{c}$) include the system call number, syscall flags and modes, the union of which determines the system call functionality. These constant values can be different across ABIs and platforms. For example, the `sys_read` system call has system call number 0 on x86-64 platforms and 3 on i386 platforms. Directly comparing these constants, as traditional NVX systems do, will cause a false alarm even if they are "semantically" equivalent.

Rule 1 resolves this issue, by translating these configuration parameters to a canonical representation before comparing them. PISC compiles this rule automatically by reading the system call tables on the fly and replace system call numbers, with their corresponding system call name before comparison. For flags, modes and any other configuration constant defined as a macro inside glibc, PISC follows the same principle. This is a fully automated procedure and thus allows DMON to seamlessly extend to additional platforms.

Rule 2 - Struct Layout Canonicalization. Data parameters $\boldsymbol{p}$ of a system call may include some struct type parameters. Determining equivalence of struct type parameters is challenging because C structs are not necessarily bit-for-bit identical across ABIs, even when the arguments are semantically equivalent; different platforms define different packing (i.e., padding) and alignment rules for a data structure. To allow for bitwise comparisons of such structs, PISC canonicalizes structs to an internal "shadow" type that uses fixed size fields and is carefully constructed so it has the same layout across platforms. Again, this procedure is completely automated and thus extensible to new architectures.

Rule 3 - Implicit Parameter Canonicalization. Beyond differences in the syscall numbers for the same system call, heterogeneous-ISA variants may use similar yet different system calls for the same functionality, because not all system calls are available on every platform. According to our definition of semantic equivalence of system calls, such similar system calls represent an equivalent functionality F. Checking equivalence of the data parameters $\boldsymbol{p}$ in this case serves as a key to determine semantic equivalence of these system calls.

x86-64 kernels, for example, implement both `sys_open` and `sys_openat`. ARMv8 kernels, on the other hand, do not implement `sys_open`. ARMv8 variants therefore always use `sys_openat` to open a file. `sys_openat` is similar to `sys_open`, but has an additional argument that can hold the file descriptor of a directory. If the `pathname` argument of the `sys_openat` is relative, then it is interpreted relative to the directory specified in the additional argument. In this concrete example, PISC maps `sys_open` and `sys_openat` to the same system call functionality and it fully resolves equivalence of the data parameters including the directory paths that the variants are trying to access.

4.2 Distributed Monitor Design

Prior work often used a central monitor process which simultaneously supervised all of the variants [5,39,53]. Subsequent research showed that this centralized model was overly focused on simplicity and security at the expense of performance, and suggested various designs in which each variant was supervised by a dedicated monitor instance [21,26,31,52,54,55]. This dedicated monitor instance could be loaded directly into the variants' address spaces, thereby trading off the isolation between the variants and the monitor for reduced variant-monitor communication overhead. DMON combines elements of both designs. Since we run ISA-heterogeneous variants on different machines, we cannot use a central monitor that attaches locally to all variants. Instead, we use a dedicated monitor for each variant and run the monitor on the same machine as the variant it supervises. Our design does, however, enforce strict isolation between the variant and its monitor by running the monitor as a separate process that attaches to the variant using the `ptrace` API.

4.3 Inter-Monitor Communication

F-MON and L-MON communicate whenever the variants execute a system call. This exchange may include system call numbers, serialized system call arguments, system call results, or instructions on how to proceed from a system call entry point (see Sect. 4). In many cases, particularly when the system call being executed is deemed security-sensitive, communication must happen synchronously. For instance, L-MON cannot allow the leader variant to proceed past a system call entry point until all instances of F-MON have serialized the state of their corresponding variant, and until they have sent this state to L-MON. F-MON needs to wait even longer as it cannot allow the follower variants to proceed until L-MON has compared the variant states and it has received L-MON's confirmation that the states match. To achieve good performance, DMON therefore requires a reliable inter-monitor communication channel with minimal latency and high bandwidth. We experimented with various designs of this communication channel and implemented them in our RC-COM, which exposes the inter-monitor communication API to our monitors.

Network Protocol Choice. The most obvious protocol that meets our reliability demands is TCP, which we used as the basis for our first implementation of RC-COM. However, even with extensive tuning, our TCP-based implementation had poor throughput and high latency. As an alternative, we therefore used ENet, a lightweight UDP-based protocol that also offers reliable in-order and error-free data transfer [17]. Besides the networking hardware, the operating system also affects the communication bandwidth and latency. When a network adapter receives a packet, for example, the OS first stores the packet in a kernel-space buffer, before copying it into the receiving application's memory and transferring control to the application. Remote Direct Memory Access (RDMA) avoids these extra copy operations by allowing two communicating peers to read or write directly from or to the other peer's application memory, thus bypassing the kernel's networking stack. We implemented an RDMA-based version of our RC-COM using Mellanox ConnectX 100 gigabit Ethernet interfaces [33] and the Mellanox Messaging Accelerator user-space networking library [29].

4.4 Optimizations

To further improve DMON's performance, we implemented several optimizations that reduce the number of the data packets exchanged by our monitors.

Permissive Filesystem Access. Traditional NVX systems allow one variant to perform *all* I/O operations and then replicate the results to the other variants. Even though this replication mechanism seamlessly provides identical inputs to all variants, it is not always necessary. Specifically, there is no need to replicate read accesses to read-only files that were identical on all physical machines when DMON started, as long as the files have not been modified while DMON was running. We refer to such files as *static files* and designed the replication handlers for read-only operations such as `sys_read` and `sys_fstat` so that all variants may (optionally) read static files directly from their local file system, thus bypassing the I/O replication. For this optional optimization, DMON requires that the application's root directory has the same path name on all machines as well as identical content including sub-directories with the exception of executables and shared libraries.

Asynchronous Cross-Checking. Our basic approach described in Sect. 4 adds considerable overhead to every system call invocation as every cross-check happens synchronously and requires at least two network round-trips; one for F-MONs to send the system call states of their supervised variants to L-MON, and one for L-MON to instruct F-MONs on how to proceed (abort or continue execution of the variant). We developed a technique which we call *asynchronous cross-checking* to reduce this overhead. Inspired by previous work [26, 52], the idea is to classify system calls into three categories—highly sensitive, moderately sensitive, and non-sensitive—based on the system call number and/or arguments. With asynchronous cross-checking, highly sensitive system calls still execute in lock-step, as before. When F-MON deems a system call moderately sensitive, however, it still sends the system call state information to L-MON, but then immediately resumes execution of the supervised variant without waiting for a reply from L-MON. L-MON eventually receives the state information and may detect a divergence. In that case, L-MON will instruct F-MONs to abort execution through a separate error channel that is used only for this specific purpose. Non-sensitive system calls can execute without any cross-checking.

5 Implementation

We implemented DMON for GNU/Linux. DMON runs natively on the x86-64 and ARMv8 architectures. DMON also has partial support for ARMv7 and i386. Currently, our prototype has 35k lines of C and C++ code and supports variants compiled with the stock versions of gcc and Clang. We do, however, require the variants to link against our patched C library (see *Virtual System Calls* below for details). DMON currently supports 100 system calls. Adding support for additional system calls generally requires a trivial amount of engineering effort (typically less than 10 lines of code), as DMON defines helper macros to replicate and cross-check most types of system call arguments (see Sect. 4.1). Our helper macros resemble those used in ReMon [52], but differ from them as they automatically apply PISC, thus making our macros fully portable. The type of cross-checking depends on the security-sensitivity of the call (see Sect. 4.4).

DMON always cross-checks highly sensitive system calls in lock-step. Moderately sensitive calls are checked asynchronously. Non-sensitive calls are not checked at all. The type of replication depends on the kind of results the system call returns. DMON enforces replication for all I/O operations that are not reads from static files (see Sect. 4.4), and for all system calls that return mutable program state. Read operations from static files execute without replication if the permissive filesystem access optimization is enabled. System calls that must be executed by all variants are not subject to any replication.

Virtual System Calls. On most architectures, Linux loads a Virtual Dynamic Shared Object (VDSO) or vsyscall page into the address spaces of all user-space programs. These executable code pages expose virtual system calls, which allow the program to execute certain system calls (e.g., `sys_gettimeofday`) without switching into kernel space. Most NVX systems either hide, replace, or disable the VDSO and vsyscall page because virtual system calls are invisible to the monitor. For our prototype, we patched the C library our variants link against so that virtual system calls are disabled.

6 Security Analysis

Scope. NVX systems can prevent usage of *absolute* code addresses by adopting Address Space Partitioning (ASP) [12,31,50] that lays out the variants' code sections to have non-overlapping/disjoint virtual addresses. In this Section, we focus on evaluating the additional security DMON can provide through ISA/ABI-heterogeneity. Specifically, we show the extent to which ISA/ABI-heterogeneity prevents concrete code-reuse and data-only attacks that cannot be easily stopped using existing NVX systems.

Analysis Targets and Configurations. We used four popular server applications—Nginx 1.14.2, Lighttpd 1.4.52, Redis 5.0.1, and ProFTPD 1.3.0—as our analysis targets, which is in line with previous work on security-oriented NVX systems [26,31,52,54]. We evaluated the security of a heterogeneous configuration with one program variant compiled for Intel x86-64 and one for ARMv7.

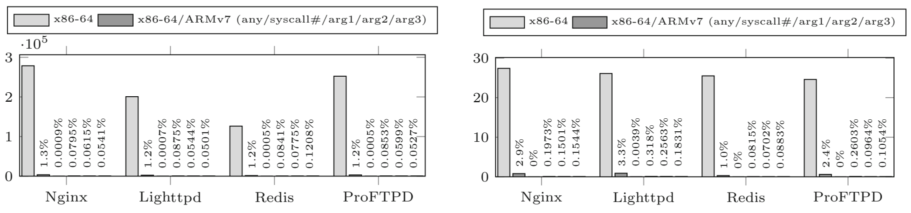
6.1 Code Layout Diversity

Existing NVX systems that deploy address space partitioning (ASP) can be bypassed using attacks that rely on partial overwrites of code pointers such as return addresses or function pointers [13,16]. The basic idea is to force the program to produce a (number of) legal code pointer(s) at memory locations that the attacker can overwrite. The attacker then overwrites the least significant bits or adds arbitrary offsets to each of these code pointers, and thereby diverts the execution of the program to a series of attacker-chosen gadgets (i.e., instruction sequences ending with indirect branches, such as return instructions). In the PIROP attack, for example, Goktas et al. exploited a vulnerability in the Asterisk communication server that allowed them to produce legal return addresses at an attacker-controlled position on the stack [16]. They then overwrote the least significant byte of each of these return addresses to build a PIROP gadget chain, which they then invoked by exploiting another vulnerability.

These attacks can in principal bypass existing NVX systems because they do not require any information leakage (which the NVX system would detect), and because the same partial pointer overwrites can achieve the same results in each variant. In this section, we show that DMON makes these position-independent code-reuse attacks far more challenging because ISA/ABI-heterogeneity substantially reduces the number of position-independent gadgets available to the attacker.

Position-Independent Gadget Availability. Position-independent gadgets are instruction sequences that can be *reliably* invoked by patching legal code pointers. We consider two ways to patch legal code pointers. First, an attacker could overwrite an offset variable that is later added to a code pointer in a pointer arithmetic operation. This primitive allows attackers to reliably invoke any gadget, as long as the internal layout of the target binary is known. Second, the attacker could overwrite the least significant bits of a code pointer directly using a memory write vulnerability. This primitive is far less potent than the former, as it allows the attacker to overwrite only the 8 least significant bits (i.e., one byte). Overwriting more than one byte is not possible unless the attacker knows the base address of the target binary because the ASP scheme randomizes all but the 12 least significant bits of each base address.

We compiled a list of the position-independent gadgets in both our x86-64 and ARMv7 binaries as follows. We first collected the addresses of (i) all instructions that immediately follow call instructions, and (ii) all address-taken functions in the program. The former is an approximation of the set of legal return addresses that could exist in the program's address space at any given point during its execution. The latter is the set of other code pointers that could be found in the program's memory. Combined, this list approximates the set of pointers that *could* potentially be patched by attackers to construct position-independent code-reuse payloads. We then used Ropper to generate lists of regular ROP gadgets consisting of 15 instructions or less [40]. This, again, is consistent with related work [16]. Next, we combined the two lists for each binary as follows. For



(a) Offset overwrite (b) Partial pointer overwrite

Fig. 2. Number of position-independent code-reuse gadgets.

every code pointer in the first list, we calculated the (i) addresses of all gadgets relative to the pointer, and (ii) absolute addresses of gadgets that only differ from the code pointer in their 8 least significant bits. The former is the set of gadgets reachable through offset overwrites, while the latter is the set of gadgets reachable through partial pointer overwrites.

Next, we correlated the position-independent gadgets found for the x86-64 binary with those found for ARMv7. For each x86-64 gadget, we checked whether there is an ARMv7 gadget that can be reached using the same offset overwrite/partial pointer overwrite. We then eliminated gadgets whose absolute address or offset from the source code pointer is not 4-byte aligned, since code pointers patched in either way would be unaligned on ARMv7 and would trigger an unaligned instruction exception when the gadget is invoked. We collected 2553 code pointers from Nginx, 1988 code pointers from Lighttpd, 1732 code pointers from Redis, and 4514 code pointers from ProFTPD. Figure 2 shows how many gadgets can be reached on average from each code pointer by offset overwrite and partial pointer overwrite attacks. In a traditional NVX system where all variants are compiled for Intel x86-64, all of the gadgets identified in the x86-64 binary would survive. In contrast, in all four of our target programs, and for both code pointer patching strategies, less than 3.3% of the gadgets survive in an NVX configuration with a x86-64 variant and an ARMv7 variant.

Position-Independent Gadget Semantics. The final step of an exploit is often to call a security-sensitive function or a system call with attacker-specified arguments (e.g., `execve` with "`/bin/sh`" as argument for a shell). The ABI-heterogeneity provided by DMON imposes another constraint on chaining gadgets to build such an exploit. Because different architectures have different calling conventions for system calls and subroutines, as shown in Table 1, the attacker should chain a sequence of gadgets that prepare the same set of arguments, but in a different way for each architecture. For example, in an ARMv7 variant, the attacker must use `r7` to prepare a system call number, whereas in a x86-64 variant the same attacker must use `rax`. To show the difficulty of constructing a code-reuse attack that performs one or more system calls and/or subroutine calls, we analyzed the semantics of position-independent gadgets surviving under DMON. Specifically, we looked for gadgets that read a value from memory and write that value into the system call number register, or the registers for one of the first three arguments of a system or function call. As shown in Fig. 2, only a small fraction of the position-independent gadgets have suitable semantics for argument preparation (see 3rd to 6th bars in the figure). More interestingly, system call number preparation gadgets are rare compared to other argument preparation gadgets. In a standalone ARMv7 binary of Nginx, Redis, and ProFTPD, we could not find a single partial-pointer-overwrite based position-independent gadget which can load a system call number. Obviously then, we also could not find such gadgets among those that survive across architectures.

Table 1. Comparison of function and syscall conventions.

arch/ABI	syscall#	arg1	arg2	arg3	arg4	arg5	arg6	arg7	result
x86-64	–	rdi	rsi	rdx	rcx	r8	r9	–	rax
arm/EABI	–	r0	r1	r2	r3	Stack	Stack	Stack	r0-r3
x86-64	rax	rdi	rsi	rdx	r10	r8	r9	–	rax
arm/EABI	r7	r0	r1	r2	r3	r4	r5	r6	r0

Table 2. Number of diversified data structures.

	Artificial	DMON	Total
Nginx 1.14.2	53	335	365
Lighttpd 1.4.52	15	95	116
Redis 5.0.1	57	158	209
ProFTPD 1.3.0	23	72	84

6.2 Structure Layout Diversity

Apart from code layout diversity we achieve from ISA-heterogeneity, DMON naturally provides data structure layout diversity. Due to differences in sizes of pointers and primitive data types, as well as differences in struct packing and alignment, data structures rarely have the same sizes and layouts across platforms. Diversifying structure layouts greatly raises the bar for attacks that require knowledge about data structure definitions including certain types of data-only attacks that rely on deterministic placement of structure fields [15,23].

Previous NVX systems could achieve structure layout diversity by artificially reorganizing structures at compile time. However, in practice, only a limited number of structs can be diversified at compile time. Specifically, it is not safe to diversify (i) structures used as arguments or return types of external library functions, (ii) structures with an initialization list, (iii) structs cast to different types, etc. [10,30]. We implemented existing type-based structure layout randomization techniques [10,30], and we examined struct layouts in a set of server applications to show how much structure layout diversity DMON can naturally achieve, compared to the number of structures that can be artificially diversified. As shown in Table 2, our heterogeneous NVX system provides a much higher degree of structure layout diversity than one can achieve using a compiler-based technique.

Case Study: ProFTPD SSL Private Key Leak. Hu et al. demonstrated an information disclosure attack on ProFTPD, in which the attacker locates a base pointer to an SSL context data structure, and then uses Data-Oriented Programming (DOP) gadgets to traverse through the context and 6 other data structures, ultimately reaching a private key, which is then leaked to a remote attacker [23]. DMON can prevent this attack because the layouts of the 6 data structures differ across architectures. We examined the relevant data structures in ARMv7 and x86-64 binaries of ProFTPD and found that 4 of the 6 pointer fields that need to be dereferenced in this attack are located at different offsets in the two binaries. A DOP exploit that traverses through the structs therefore cannot simultaneously reach and leak the private key on both platforms without triggering a divergence in DMON.

7 Performance Evaluation

We conducted an extensive performance evaluation of DMON using handwritten microbenchmarks (see Sect. 7.1), as well as popular high-performance server applications (see Sect. 7.2). We tried two different configurations:
The ***low-end configuration*** had an ARMv8 variant running on a Raspberry Pi 3 Model B board with a quad-core 1.2GHz Broadcom BCM2837 64-bit CPU and 1GB of RAM, running the 64-bit ARM Debian 9 distribution of GNU/Linux, as well as an x86-64 variant running on a desktop machine with a quad-core Intel i5-6500 CPU and 16 GB of RAM, running the x86-64 version of Ubuntu 16.04.5 LTS. The machines were connected through a *private* 100 megabit Ethernet connection with approximately 0.5 ms latency.

The ***high-end configuration*** had an x86-64 variant running on a desktop machine with an octa-core Intel i9-9900K CPU and 32 GB of RAM, and an x86-64 variant running on a machine with a quad-core Intel i5-6500 CPU and 16 GB of RAM. Both machines ran the x86-64 version of Ubuntu 16.04.5 LTS and were connected using a private 100 gigabit connection between two Mellanox ConnectX Ethernet interface cards. These RDMA-capable cards support the Mellanox Messaging Accelerator, a user-space networking library with low latency.

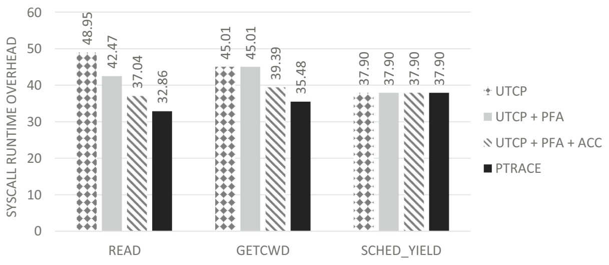
In both configurations, we ran the leader variant on the slower machine. We evaluated two implementations of RC-COM (see Sect. 4.3) for the low-end configuration. The first implementation, which appears as KTCP in the graphs, uses standard TCP/IP. The second implementation uses the ENet protocol. For the high-end configuration, we additionally evaluated an implementation that leverages the Mellanox Messaging Accelerator. This implementation appears as UTCP (short for user-space TCP) in the graphs. We could not test this UTCP implementation for low-end configuration as it was not supported by our ARMv8 board. Finally, we evaluated the impact of our replication and cross-checking optimizations described in Sect. 4.4. Our Asynchronous Cross-Checking and Permissive Filesystem Access optimizations appear as ACC and PFA respectively in the graphs.

7.1 Microbenchmarks

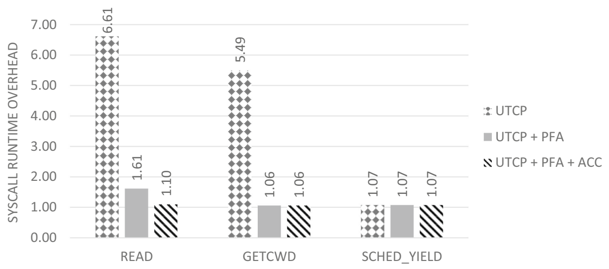
To measure the overhead introduced by DMON, we designed microbenchmarks to test our optimizations (see Sect. 5). We used the following system calls:

1. **sys_read(STATIC_FILE_FD, buf, 512)** is treated as a moderately sensitive system call. As such, this microbenchmark benefits from our asynchronous cross-checking optimization and skips replication if all optimizations are enabled (see Sect. 4.4).
2. **sys_getcwd(buf, 512)** The results of this system call do not need to be replicated, as long as the current working directory is either the application's root directory, or one of its subdirectories (see Sect. 4.4).
3. **sys_sched_yield()** is a representative of system calls that require neither cross-checking nor replication.

Figure 3(a) shows the execution time under DMON's hign-end configuration relative to the native execution time. We used our UTCP implementation of RC-COM for all experiments, but did run separate tests with and without our permissive file access (PFA) and asynchronous cross-checking (ACC) optimizations. We also measured the execution time without cross-checking and replication (PTRACE). This experiment shows that the `ptrace` API is the main performance bottleneck in our system. Prior work illustrates that replacing `ptrace`-based monitoring by in-process alternatives allows for a much wider range of security-performance trade-offs [26,52].



(a) Fully distributed ptrace-based NVX

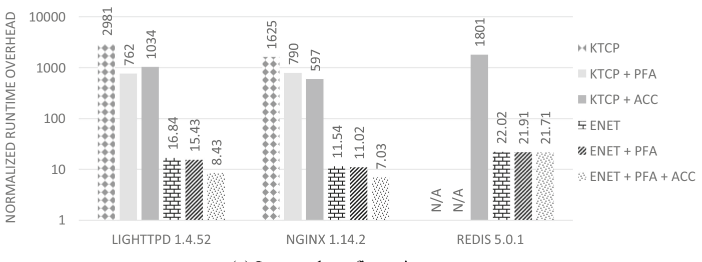


(b) Proof of concept distributed in-process NVX

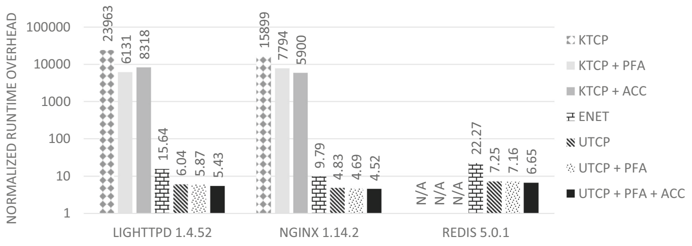
Fig. 3. Microbenchmarks for *high-end* configuration

The results show that the overhead can be attributed to the network communication of our replication and cross-checking mechanisms, and the context switching caused by `ptrace`. PFA reduces the overhead of `read` from 48.95× to

42.47×, but does not affect the other benchmarks. ACC further decreases overhead of `read` and `getcwd`, from 42.47× to 37.04× and from 45.01× to 39.39× respectively. `sched_yield`'s performance is unaffected, since DMON does not perform any cross-checking for this system call. Finally, the rightmost columns in Fig. 3(a) indicate that the context switching overhead of `ptrace` is by far the biggest contributor to DMON's overhead. We hypothesized that monitoring non-sensitive system calls in-process, as was done in prior work [21,38,52], would substantially reduce the context switching overhead, and set up an experiment to confirm this hypothesis. Specifically, we implemented a distributed in-process NVX system using the `syscall_intercept` [45], and evaluated it on the same microbenchmarks we used for the `ptrace`-based prototype. Our in-process prototype implements the optimizations described in Sect. 4.4, but only supports a small set of system calls. Figure 3(b) shows that in-process monitoring reduces the per-system call overhead from 32.86–37.90× to only 6–10% with all optimizations enabled.



(a) Low-end configuration



(b) High-end configuration

Fig. 4. Server benchmarks in two configurations

7.2 Server Benchmarks

We evaluated DMON on 3 popular server applications—Nginx 1.14.2, Lighttpd 1.4.52 and Redis 5.0.1—that were also used to evaluate prior work [21,26,31,52]. For each of our experiments, we connected a benchmarking client to the leader machine through a 100 megabit Ethernet connection (for our low-end configuration) or a 1 gigabit Ethernet connection (for the high-end configuration). Figure 4 shows our results. We used the `wrk` client to evaluate Nginx and Lighttpd, and the `redis-benchmark` utility to evaluate Redis. We configured `wrk` to repeatedly request the same static 4KB web page for 10 s using 10 parallel connections, and `redis-benchmark` to simulate 50 clients issuing 100,000 requests in total. Running `redis-benchmark` under DMON's slowest configurations would take over a day, so we skipped them and denote it as *N/A* in Fig. 4. The latency on the 100 megabit link was just under 0.5 ms, whereas the latency on the 1 gigabit link was under 0.1 ms. With all of DMON's optimizations enabled, the performance overheads ranged between 7.03× and 21.71× for the low-end configuration, and between 4.52× and 6.65× for the high-end configuration.

7.3 Comparison with Other NVX Systems

We compared the performance of DMON with traditional NVX systems. Thanks to our inter-monitor communication optimizations, DMON achieves similar (or better) performance than traditional single-host NVX systems with `ptrace`-based monitors. Specifically, GHUMVEE (the state of the art `ptrace`-based NVX system) was tested on the same server applications (albeit slightly older versions), and in highly similar conditions, with a 1 gigabit link that had less than 0.1 ms of latency. GHUMVEE's overhead on Lighttpd was 7.0× on a saturated server (vs 5.43× for DMON), and 12.48× for Redis (vs 6.65× for DMON) [52]. Delegating the monitoring of non-sensitive system calls to an in-process monitor would substantially reduce the overhead, as was shown in prior work [21,38,52], as well as in Sect. 7.1. We summarize our findings in Table 3. DMON (IP) refers to our PoC distributed in-process prototype and DMON (CP) refers to our distributed `ptrace`-based implementation. As GHUMVEE was not evaluated on microbenchmarks, and DMON (IP) currently does not support server applications, these numbers are shown as *N/A* in the table.

Table 3. Comparison with other NVX systems.

NVX system	Monitor type	Distributed	Overhead	
			System call	Server apps
GHUMVEE [52]	CP	No	N/A	7.0–12.48×
DMON (CP)	CP	Yes	**32.86–37.90**×	**4.52–6.65**×
Varan [21]	IP	No	36–139%	11–37%
DMON (IP)	IP	Yes	**6–61%**	**N/A**

8 Discussion

Performance Improvements. While developing DMON, we identified a promising path to substantially improve our monitoring performance. We could replace our `ptrace`-based monitoring mechanism with an in-process alternative based on API call interception [21], or hardware-based virtualization extensions [26]. Securing an in-process monitoring design is challenging, however.

Leveraging Hardware Features. A potential advantage of running variants on different architectures is that the NVX system could leverage hardware security features available on one platform to protect software running on other platforms. A hypothetical configuration in which DMON runs one variant on an ARMv8.5-A CPU and one variant on an Intel x86-64 CPU could be used to bring the benefits of memory tagging to Intel x86-64 software.

Micro-Architectural Attacks. While our primary focus was on defending against memory exploits, we believe DMON might also be able to stop certain micro-architectural attacks. Rowhammer attacks in particular would become exceedingly hard to launch against DMON [19,41,49]. To build reliable Rowhammer attacks, the attacker needs to know exactly how the memory controller translates physical memory addresses into DRAM addresses [37,47]. Translation schemes differ greatly across platforms, however, which makes Rowhammer attack payloads non-portable.

9 Related Work

N-Variant eXecution. Inspired by Chen and Avizienis' seminal work on N-Version Programming [2,8], Berger and Zorn proposed a system for probabilistic memory safety that could simultaneously execute identical variants with differently seeded randomizing memory allocators [3]. This system only supported trivial applications, however. Cox et al.'s N-Variant Systems monitored a much wider array of system calls, thus supporting variants of complex applications [12]. Subsequent publications explored consistent delivery of asynchronous signals [5,39], dealing with shared memory [5], thread synchronization [51], or address-dependent behavior [53], and new schemes for generating software variants [26,31,50,54]. Other researchers suggested to use NVX systems for live patch testing [20,21,25,27,32,38].

10 Conclusion

We presented DMON, a novel, distributed N-Variant Execution system that leverages diversity in ISAs and ABIs to protect against memory corruption attacks. To bypass DMON, attackers must provide exploits that simultaneously work on different platforms. DMON's heterogeneous platform setting naturally provides code layout diversity which greatly raises the bar for code-reuse attacks, and it naturally provides a higher level of structure layout diversity than what existing compiler-based techniques can provide. To avoid benign divergences caused by expected cross-platform differences, we propose PISC, a technique that transforms system call states into platform-independent states. Also, we introduce new optimization strategies to alleviate performance issues that are unique to the distributed NVX setting. Our performance evaluation shows that the proposed optimizations, combined with an optimized network protocol, greatly reduce the performance overhead without sacrificing DMON's security.

Acknowledgments. The authors thank Kostis Kaffes, Marios Pomonis, Georgios Detorakis, Lefteris Kokoris-Kogias, Anil Altinay, Mohaned Qunaibit, Paul Kirth, David Gens, Adrian Dabrowski and our reviewers. This material is based upon work partially supported by the Defense Advanced Research Projects Agency under contract FA8750-16-C-0260, by the United States Office of Naval Research under contract N00014-17-1-2782, and by the National Science Foundation under award CNS-161921. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency or its Contracting Agents, the Office of Naval Research or its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: CCS (2005)
2. Avizienis, A.: The N-version approach to fault-tolerant software. IEEE TSE (12), 1491–1501 (1985)
3. Berger, E.D., Zorn, B.G.: Diehard: probabilistic memory safety for unsafe languages. In: PLDI (2006)
4. Bittau, A., Belay, A., Mashtizadeh, A., Mazières, D., Boneh, D.: Hacking blind. In: IEEE S&P (2014)
5. Bruschi, D., Cavallaro, L., Lanzi, A.: Diversified process replicæ for defeating memory error exploits. In: IEEE IPCCC (2007)
6. Burow, N., et al.: Control-flow integrity: precision, security, and performance. ACM Comput. Surv. (CSUR) **50**(1), 16 (2017)
7. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: CCS (2010)
8. Chen, L., Avizienis, A.: N-version programming: a fault-tolerance approach to reliability of software operation. In: FTCS (1978)
9. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: USENIX Security Symposium (2005)
10. Chen, Z., Han, H.: Attack mitigation by data structure randomization. In: Cuppens, F., Wang, L., Cuppens-Boulahia, N., Tawbi, N., Garcia-Alfaro, J. (eds.) FPS 2016. LNCS, vol. 10128, pp. 85–93. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-51966-1_6
11. Conti, M., et al.: Losing control: on the effectiveness of control-flow integrity under stack attacks. In: CCS (2015)
12. Cox, B., et al.: N-variant systems: a secretless framework for security through diversity. In: USENIX Security Symposium (2006)
13. Durden, T.: Bypassing PaX ASLR protection. Phrack Mag. **11** (2002)
14. Evans, I., et al.: Control jujutsu: on the weaknesses of fine-grained control flow integrity. In: CCS (2015)
15. Gil, R., Okhravi, H., Shrobe, H.: There's a hole in the bottom of the C: on the effectiveness of allocation protection. In: IEEE SecDev (2018)
16. Göktas, E., et al.: Position-independent code reuse: on the effectiveness of ASLR in the absence of information disclosure. In: IEEE EuroS&P (2018)
17. ENet: Reliable UDP networking library. http://enet.bespin.org
18. Gras, B., Razavi, K., Bosman, E., Bos, H., Giuffrida, C.: ASLR on the line: practical cache attacks on the MMU. In: NDSS (2017)

19. Gruss, D., Maurice, C., Mangard, S.: Rowhammer.js: a remote software-induced fault attack in javascript. In: Caballero, J., Zurutuza, U., Rodríguez, R.J. (eds.) DIMVA 2016. LNCS, vol. 9721, pp. 300–321. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40667-1_15
20. Hosek, P., Cadar, C.: Safe software updates via multi-version execution. In: ICSE (2013)
21. Hosek, P., Cadar, C.: Varan the unbelievable: an efficient n-version execution framework. In: ASPLOS (2015)
22. Hu, H., Chua, Z.L., Adrian, S., Saxena, P., Liang, Z.: Automatic generation of data-oriented exploits. In: USENIX Security Symposium (2015)
23. Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z.: Data-oriented programming: on the expressiveness of non-control data attacks. In: IEEE S&P (2016)
24. Hund, R., Willems, C., Holz, T.: Practical timing side channel attacks against kernel space ASLR. In: IEEE S&P (2013)
25. Kim, D., Kwon, Y., Sumner, W.N., Zhang, X., Xu, D.: Dual execution for on the fly fine grained execution comparison. In: ASPLOS (2015)
26. Koning, K., Bos, H., Giuffrida, C.: Secure and efficient multi-variant execution using hardware-assisted process virtualization. In: DSN (2016)
27. Kwon, Y., et al.: LDX: causality inference by lightweight dual execution. In: ASPLOS (2016)
28. Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: SoK: automated software diversity. In: IEEE S&P (2014)
29. Mellanox's Messaging Accelerator. https://github.com/Mellanox/libvma/
30. Lin, Z., Riley, R.D., Xu, D.: Polymorphing software by randomizing data structure layout. In: Flegel, U., Bruschi, D. (eds.) DIMVA 2009. LNCS, vol. 5587, pp. 107–126. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02918-9_7
31. Lu, K., Xu, M., Song, C., Kim, T., Lee, W.: Stopping memory disclosures via diversification and replicated execution. In: IEEE TDSC (2018)
32. Maurer, M., Brumley, D.: TACHYON: tandem execution for efficient live patch testing. In: USENIX Security Symposium (2012)
33. Mellanox ConnectX-5 EN Adapter Supporting 100 Gb/s Ethernet
34. Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: SoftBound: highly compatible and complete spatial memory safety for C. In: PLDI (2009)
35. Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: CETS: compiler enforced temporal safety for C. In: ISMM (2010)
36. Novark, G., Berger, E.D.: DieHarder: securing the heap. In: CCS (2010)
37. Pessl, P., Gruss, D., Maurice, C., Schwarz, M., Mangard, S.: Drama: Exploiting dram addressing for cross-cpu attacks. In: USENIX Security Symposium (2016)
38. Pina, L., Andronidis, A., Hicks, M., Cadar, C.: Mvedsua: higher availability dynamic software updates via multi-version execution. In: ASPLOS (2019)
39. Salamat, B., Jackson, T., Gal, A., Franz, M.: Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In: EuroSys (2009)
40. Schirra, S.: Ropper (2014). https://github.com/sashs/Ropper
41. Seaborn, M., Dullien, T.: Exploiting the dram rowhammer bug to gain kernel privileges. In: BlackHat USA (2015)
42. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: CCS (2007)

43. Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.: Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization. In: IEEE S&P (2013)
44. Song, D., et al.: SoK: sanitizing for security. In: IEEE S&P (2019)
45. System call intercepting library. https://github.com/pmem/syscall_intercept
46. Szekeres, L., Payer, M., Wei, T., Song, D.: SoK: eternal war in memory. In: IEEE S&P (2013)
47. Tatar, A., Giuffrida, C., Bos, H., Razavi, K.: Defeating software mitigations against rowhammer: a surgical precision hammer. In: RAID (2018)
48. van der Veen, V., Andriesse, D., Stamatogiannakis, M., Chen, X., Bos, H., Giuffrida, C.: The dynamics of innocent flesh on the bone: Code reuse ten years later. In: CCS (2017)
49. Van Der Veen, V., et al.: Drammer: deterministic rowhammer attacks on mobile platforms. In: CCS (2016)
50. Volckaert, S., Coppens, B., De Sutter, B.: Cloning your gadgets: complete ROP attack immunity with multi-variant execution. IEEE TDSC **13**(4), 437–450 (2016)
51. Volckaert, S., Coppens, B., De Sutter, B., De Bosschere, K., Larsen, P., Franz, M.: Taming parallelism in a multi-variant execution environment. In: EuroSys (2017)
52. Volckaert, S., et al.: Secure and efficient application monitoring and replication. In: USENIX ATC (2016)
53. Volckaert, S., De Sutter, B., De Baets, T., De Bosschere, K.: GHUMVEE: efficient, effective, and flexible replication. In: FPS (2012)
54. Xu, M., Lu, K., Kim, T., Lee, W.: Bunshin: compositing security mechanisms through diversification. In: USENIX ATC (2017)
55. Österlund, S., Koning, K., Olivier, P., Barbalace, A., Bos, H., Giuffrida, C.: kMVX: detecting kernel information leaks with multi-variant execution. In: ASPLOS (2019)