VARAN the Unbelievable

An Efficient N-version Execution Framework

Petr Hosek Cristian Cadar

Department of Computing
Imperial College London
{p.hosek, c.cadar}@imperial.ac.uk

Abstract

With the widespread availability of multi-core processors, running multiple diversified variants or several different versions of an application in parallel is becoming a viable approach for increasing the reliability and security of software systems. The key component of such N-version execution (NVX) systems is a runtime monitor that enables the execution of multiple versions in parallel.

Unfortunately, existing monitors impose either a large performance overhead or rely on intrusive kernel-level changes. Moreover, none of the existing solutions scales well with the number of versions, since the runtime monitor acts as a performance bottleneck.

In this paper, we introduce VARAN, an NVX framework that combines selective binary rewriting with a novel event-streaming architecture to significantly reduce performance overhead and scale well with the number of versions, without relying on intrusive kernel modifications.

Our evaluation shows that VARAN can run NVX systems based on popular C10k network servers with only a modest performance overhead, and can be effectively used to increase software reliability using techniques such as transparent failover, live sanitization and multi-revision execution.

Categories and Subject Descriptors D.4.5 [*Operating Systems*]: Reliability—Fault-tolerance

General Terms Reliability, Performance

Keywords N-version execution; selective binary rewriting; event streaming; transparent failover; multi-revision execution; live sanitization; record-replay

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey.

ACM 978-1-4503-2835-7/15/03. . . $15.00.
http://dx.doi.org/10.1145/2694344.2694390

1. Introduction

Recent years have seen a growing interest in using diversity as a way to increase the reliability and security of software systems. One form of software diversity that has attracted significant interest from the research community is the idea of running multiple diversified versions of a program in parallel in order to survive bugs and detect security attacks [5, 10, 13, 14, 40, 44, 48]. In essence, diversity can offer probabilistic guarantees that at least one variant survives a bug, or that a security attack will be flagged by divergent behaviour across variants.[1]

On the security side, these diversified variants are constructed in such a way as to reduce the probability of an attack succeeding in all of them. For example, one may generate versions with stacks growing in opposite directions [40] to prevent attacks whose success depends on the stack layout.

On the reliability side, which forms the main focus of this paper, these diversified versions are either automatically-generated variants, multiple revisions of the same application, or different programs implementing the same interface. For example, one may run in parallel multiple variants that employ complementary thread schedules to survive concurrency errors [44], multiple versions of the same software to survive update bugs [21], or multiple web browsers to benefit from the fact that many errors do not affect all browser implementations [47]. In this paper, we show that running multiple versions in parallel can be used in other reliability scenarios, such as running expensive error detectors ("sanitizers") during deployment.

To enable these scenarios, a monitor process coordinates the parallel execution of these variants and synchronises their execution, making them appear as a single application to any outside entities. While synchronisation can be performed at different levels, the most common approach is to do it at the level of system calls, for two main reasons: first, many existing diversification transformations, such as the ones discussed above, do not change the sequence of system calls (the program's *external behaviour*), and the ordering is often preserved even across different software revisions [21]. Sec-

[1] The terms *version* and *variant* are used interchangeably.

ond, system calls are the main way in which the application communicates with the outside environment, and therefore must be virtualised in order to enable the multiple versions to act as one to the outside world.

The main challenge in implementing an NVX monitor at the system call level is the trade-off between performance, security, flexibility and ease of debugging. Many implementations [7, 21, 40] use the `ptrace` mechanism offered by most UNIX-based operating systems. While easy-to-use and not requiring kernel modifications, `ptrace` is slow, and these systems see performance degradations of up to two orders of magnitude. An alternative approach is to implement the monitor in kernel space [13], which is much faster, but requires kernel patches and/or new kernel modules, and the monitor must be run in privileged mode. Furthermore, none of these approaches scales well with the number of variants (as the monitor is both a communication and synchronisation bottleneck), none are debug-friendly (`ptrace` disallows the use of *GDB*, while kernel debugging has its well-known set of limitations) and none of them have been designed to be flexible with respect to small variations in system call sequences (which can occur for certain diversification transformations and across software revisions).

In this paper, we propose VARAN,[2] a novel architecture for implementing NVX monitors. VARAN monitors operate at the system call level, run in user space (and therefore in unprivileged mode), introduce a small performance overhead for popular C10k network servers[3] and scale well with the number of versions, and provide a flexible mechanism for handling small divergences in the system call sequences issued across versions.

The rest of this paper is structured as follows. Section 2 gives a high-level overview of our approach and Section 3 presents our prototype implementation in detail. Then, Section 4 evaluates our prototype on a set of micro- and macro-benchmarks, Section 5 shows the applicability to different application scenarios, and Section 6 discusses the main implications of our design. Finally, Section 7 presents related work and Section 8 concludes.

2. Overview

Two key aspects influence the performance and flexibility of an NVX system: system call interception and version coordination. We discuss each in turn below.

2.1 System call interception

The biggest downside of existing system call monitors based on the `ptrace` interface is the high performance overhead [21, 33, 40]. For each system call performed by each version, execution must switch to the monitor process, which has to perform several additional system calls in order to copy buffers to and from the version being monitored, nullify the system call, *etc.*

For CPU-intensive applications which perform few system calls, this overhead will be amortised, translating into a modest overall slowdown. However, for heavily I/O-bound applications, the slowdown can be up to two orders of magnitude, which is unacceptable for many real-world deployments. Consequently, in order to implement a system call monitor with acceptable overhead even for heavily I/O-bound applications, we need to eliminate context switching to the monitor and back during interception and eliminate the need for additional system calls. This is accomplished through a combination of selective binary rewriting and an interprocess communication mechanism based on a fast shared memory ring buffer.

Whenever code is loaded into memory, VARAN scans each code page to selectively rewrite all system calls with jump instructions to dedicated handlers. Section 3.2 discusses in detail the main steps and challenges associated with this binary rewriting approach.

To eliminate the need for additional system calls during interception, VARAN uses a shared ring buffer to communicate between versions. This ring buffer is heavily optimised for performance: it is stored in memory, allows largely lock-free communication, and does not require the dispatch of events to different queues. These aspects are discussed in detail in Section 3.3.

2.2 Event-streaming architecture

In prior NVX systems, versions are typically run in lockstep, with a centralised monitor coordinating and virtualising their execution. Essentially, at each system call, the versions pass control to the monitor, which waits until all versions reach the same system call. Once this happens, the monitor executes the system call and communicates the result to each individual version. If two or more versions try to break the lockstep by executing different system calls, the monitor needs to either terminate the entire application or continue executing a subset of the versions in lockstep.

This approach has two key disadvantages. First, the centralised monitor is a bottleneck, which can have a significant impact on performance. Note that in addition to the synchronisation overhead, this centralised monitor makes the NVX application execute at the speed of the slowest individual version.

Second, this approach is totally inflexible to any divergence in the sequence of system calls executed across versions. This is an issue both when running automatically-diversified variants, where certain transformations may affect the external behaviour, and when running existing software revisions, where changes in the sequences of system calls can occur between revisions.

[2] VARAN's name comes from the scientific name *Varanus*, commonly known as the *monitor* lizard. Varan is also a name of the Kaiju monster that first appeared in the 1958 movie *Varan the Unbelievable*.

[3] Numeronym used for servers capable of concurrently handling ten thousand connections.

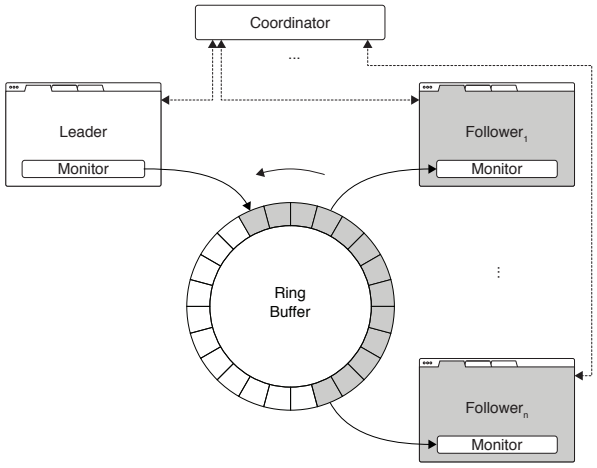

Figure 1. The event-streaming architecture of VARAN.

To address these limitations, VARAN uses a new approach which we call *event streaming*. In this decentralised architecture, depicted in Figure 1, one of the versions is designated as the *leader*, while the others are *followers*. During execution, the leader records all events into a shared ring buffer, which are later read by followers to mimic the leader's external behaviour (§3.3). Events consist primarily of regular system call invocations, but also of signals, process forks (*i.e.* `clone` and `fork` system calls) and exits (*i.e.* `exit` and `exit_group` system calls).

In general, any version can be the leader, although in some situations some may be a better choice than others–*e.g.*, when running multiple software revisions in parallel, one might prefer to designate the newest one as leader. However, the leader can be easily replaced if necessary, *e.g.*, if it crashes (§3.3.2).

The only centralised component in this architecture is the *coordinator*, whose main job is to prepare the versions for execution and establish the necessary communication channels. At a high level, the coordinator first loads the variants into memory, injects several special handlers and memory objects into their address spaces, rewrites any system calls in their code with jumps to the special handlers and then starts executing the variants (§3.1) in a decentralised manner.

2.3 Rewrite rules for system call sequences

In addition to eliminating the central monitor bottleneck, our event-streaming architecture also supports (small) divergences between the system call sequences of different variants. For example, different software revisions can be run inside a classical NVX system only as long as they all issue the same sequence of system calls [21]. However, software patches sometimes change the external behavior of an application. In particular, many divergences in system call traces fall into the following two categories: (i) *addition/removal*, characterising situations when one of the versions performs (or conversely does not perform) an additional system call, typically as a consequence of an additional check, and (ii) *coalescing*, covering the situations when a (repeated) sequence of system calls is executed a different number of times in each version (*e.g.*, one version might execute two `write` system calls, while another version executes only one `write` system call to write the same bytes because extra buffering is used).

VARAN is the first NVX system that is able to deal with such changes. When followers process the event sequence streamed by the leader, they can rewrite it to account for any such differences: *e.g.*, they can skip and merge system calls, or perform some calls themselves. We provide a flexible implementation of such rewrite rules using Berkeley Packet Filters (§3.4).

3. Prototype

We have implemented our approach in a prototype (to which we will also refer as VARAN), targeted at multi-core processors running x86-64 Linux. VARAN works on off-the-shelf binaries (both stripped and unstripped) and supports single- as well as multi-threaded applications.

When it starts, VARAN first sets up the address spaces of all program versions and establishes the needed communication channels (§3.1). It then performs selective binary rewriting to replace all system calls with jump instructions (§3.2). After these initial stages, the event streamer component of VARAN ensures the coordination of the leader and its followers (§3.3).

3.1 Setup of address spaces and communication channels

The main steps involved in the setup of version address spaces and the needed communication channels are shown in Figure 2. To run multiple versions in parallel, the user launches VARAN's *coordinator* providing the paths to all versions, together with any command line arguments required to start them (Step Ⓐ in Figure 2).

The *coordinator* first creates the shared memory segment used for communication among versions, and then spawns the *zygote* process (Ⓑ), which is responsible for starting the individual versions. The coordinator communicates with the zygote via a UNIX domain socket. For each version i that needs to be spawned, the coordinator sends a fork request to the zygote over this socket pair, which includes the path to that version executable, the command line arguments, and the end-point of a socket pair which will be used for the subsequent communication between the coordinator and that version (Ⓒ$_i$). After receiving this request, the zygote spawns a new process, which first finalises the communication with the coordinator (Ⓓ$_i$). The coordinator then sends the shared memory segment descriptor to this process, which maps it inside its address space.

In the final step, the new process starts executing inside the monitor code, which loads the specified ELF executable and sets up the initial address space as described in the ELF

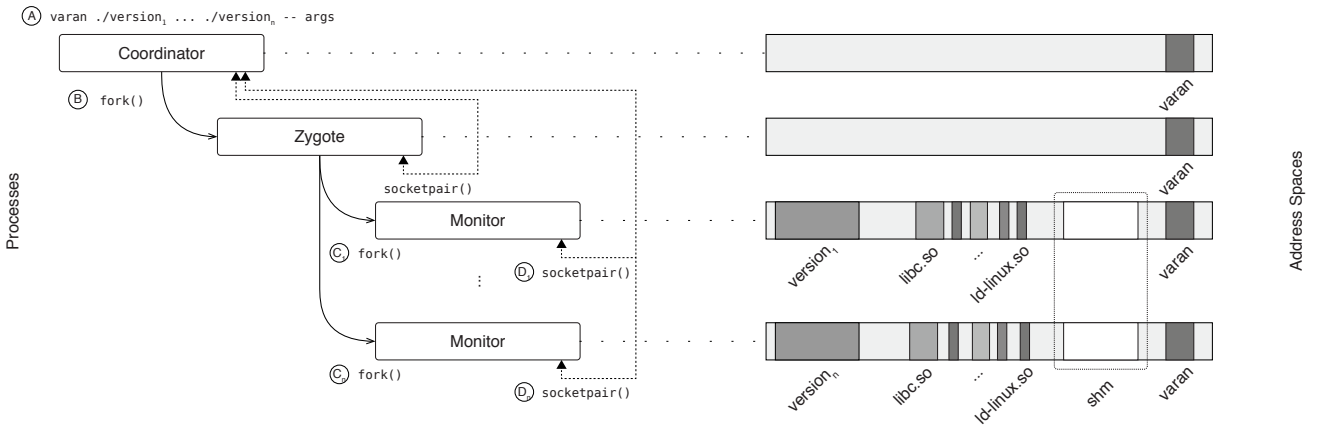

Figure 2. Setup of address spaces and communication channels.

headers. If the program requires a dynamic linker, VARAN loads the linker image specified in the header as well. The text segments of both the application and the dynamic linker are then processed by the binary rewriter (§3.2). Finally, VARAN jumps to the application entry point as specified in the ELF header, starting the execution of the application version.

The right-hand side of Figure 2 shows the address spaces of the coordinator, zygote, and program versions. When run with VARAN, program versions have two new segments mapped into their address spaces: the shared memory segment used for communication among versions ("shm") and the VARAN statically-linked library ("varan"). Note that VARAN does not prevent address-space layout randomisation schemes to be used by the operating system.

Coordinator. To set up the address spaces of the versions, the coordinator acts as a specialized preloader, inspired by *rtldi*.[4] However, the coordinator does not attempt to replace the existing dynamic linker, which would be unnecessarily complex and may affect compatibility with existing applications. Instead, it simply intercepts the system calls performed by the linker to enable the binary rewriter (§3.2) to rewrite the code of dynamically-linked shared libraries. One important advantage of our interception mechanism is that we do not make use of `ptrace` to intercept calls to the dynamic linker—instead, the binary rewriter is used to rewrite all the system calls done by the linker with jumps into the coordinator code. As a result, VARAN can be used in combination with existing `ptrace`-based tools such as *GDB* or *strace*, which greatly simplifies debugging.

Zygote. The role of the zygote is to spawn new processes on request from the coordinator. Zygote processes are already used in systems such as *Android* and *Chrome* [12]—in this paper, we use the term to refer to the architectural pattern rather than a particular implementation, as VARAN provides its own clean-slate implementation. While it would be technically possible for the coordinator to create the processes in which versions run, this would bring some complications regarding the communication channels: for example, the second version spawned would inherit the communication channel between the first version and the coordinator, which would be undesirable.

Monitor. The monitor code is built as a statically-linked, position-independent library, to make sure it does not stand in the way of any segments which have to be loaded by the application at fixed addresses. To ensure that the code can be compiled like this, we must avoid using any global variables (*i.e.* those in the `.data` section). One consequence is that VARAN cannot use any of the existing C libraries such as *GNU C Library*, as these are not typically built to support this requirement. Instead, VARAN provides its own implementation of the necessary C library functions based on the *Bionic C library*.[5] To support the use of Linux system calls, VARAN uses a modified version of the `linux_syscall_support.h` header.[6]

3.2 Binary Rewriting

To intercept system calls, VARAN uses selective binary rewriting [35]. Unlike traditional dynamic binary rewriting implemented by tools like DynamoRIO [26] or Pin [32], where the entire process image is being rewritten, often introducing a significant performance overhead, VARAN only replaces the instructions for performing system calls (*i.e.* **`int`** `$0x80` on x86 and **`syscall`** on x86-64).

The rewriting itself is done when a segment is mapped into memory with executable permissions, or an existing memory segment is marked as executable. During rewriting, VARAN scans the segment searching for system call instructions using a simple x86 disassembler. Every system call found is

[4] `http://www.bitwagon.com/rtldi/rtldi.html`

[5] `https://android.googlesource.com/platform/bionic`

[6] `https://code.google.com/p/linux-syscall-support/`

rewritten with a jump to an internal system call entry point. This process is complicated by the fact that while a system call instruction is only one byte long, a jump instruction requires five bytes. Therefore, in order to rewrite the system call with a jump, we also need to relocate some of the instructions surrounding the system call—*i.e.* perform binary detouring via trampolines [22]. On the rare occasions when this is not possible (*e.g.*, because the surrounding instructions are potential branch targets), we replace the system call with an interrupt (`INT 0x0`). This interrupt is handled by VARAN through a signal handler installed during initialisation, which redirects the control flow to the system call entry point as for other system calls.

The system call entry point first saves all registers, and then consults an internal system call table to check whether there is a handler installed for that particular system call; if so, it calls that handler, otherwise it invokes the default handler. After processing the system call, the entry point handler restores all registers and returns to the original caller (using `sigreturn` in the case of system calls intercepted via an interrupt). The system call entry point also implements support for restarting system calls (*i.e.* signaled by the `-ERESTARTSYS` error code). This is used in certain scenarios supported by VARAN such as transparent failover (§5.1).

The internal system call table can be easily changed to accommodate various application scenarios. In particular, the only difference between the leader and the followers is the system call table. For example, the `write` system call would be redirected in the leader to a function that performs the call and records its result in the shared ring buffer, while in the followers it would be redirected to a function that reads the results from the shared buffer without making the call. VARAN also provides a Python script which can produce new tables and their implementations using templates.

Finally, note that in order to prevent potential attackers to easily inject system calls into the program, the binary rewriter follows a W⊕X discipline throughout execution, making sure that segments are not marked as both writable and executable at the same time.

3.2.1 Virtual System Calls

Certain Linux system calls are accelerated through the *vsyscall* page and the *vDSO* segment. These are mapped into the address space of each Linux process, and contain system call implementations. These *virtual system calls* do not incur the context switch overhead between kernel and user space associated with standard system calls.

The *vsyscall* page was introduced first, but is being deprecated in favor of the *vDSO* segment. The main reason for this development is that the *vsyscall* page is mapped to a fixed address, making it susceptible to return-oriented programming attacks [37]. To address this issue, the vDSO segment is mapped to a random address. Since the segment is dynamically allocated, it can also support an arbitrary number of virtual system calls (currently `clock_gettime`, `getcpu`, `gettimeofday` and `time`).

Virtual system calls represents one of the major limitations of `ptrace`-based monitors. Since these system calls are entirely implemented in user space, they cannot be intercepted via `ptrace` . This is an important limitation: as these system calls provide access to timing information, they are often used as a source of non-determinism (*e.g.*, for random number generators) and their handling is critical for any NVX system.

To our knowledge, VARAN is the first NVX system which handles virtual system calls, using binary rewriting. Handling calls made via the *vsyscall* page is easier because the function symbols are always mapped to the same address. To handle *vDSO* calls, we first need to determine the base address of the *vDSO* segment; this address is passed by the kernel in the ELF auxiliary vector via the `AT_SYSINFO_EHDR` flag.[7] Second, we need to examine the ELF headers of the *vDSO* segment to find all symbols. Identifying calls to these symbols is more complicated than in the *vsyscall* case because these symbols are allocated at arbitrary addresses. Instead, we replace the entry point of each function with a jump to dynamically generated code which sets up the stack and then issues a call to the VARAN system call entry point as in the case of regular system calls. Furthermore, we provide a trampoline, which allows the invocation of the original function, by moving the first few instructions of each function to a new place, followed by a jump to the original code. This allows VARAN to take advantage of the virtual system call mechanism to further improve performance.

3.3 Event Streaming

As we discussed briefly in Section 2 and illustrated graphically in Figure 1, the leader records all external events into a shared ring buffer, while the followers replay them to mimic the leader's behavior. The leader is the only version interacting with the environment, *i.e.* executing the system calls, with the exception of system calls which are local to the process (*e.g.*, `mmap`).

As in any NVX system operating at the level of system calls, VARAN has to be aware of the system call semantics, in order to transfer the arguments and results of each system call. VARAN currently implements 86 system calls, which were all the system calls encountered across our benchmarks.[8]

3.3.1 Shared ring buffer

For fast communication, the leader and its followers share a common ring buffer of fixed size, which is held entirely in memory. Our initial solution used a separate shared queue for each process [18, 29], with the coordinator acting as an event pump—reading events from the leader's queue and dispatching them into followers' queues. This approach

[7] `https://www.gnu.org/software/libc/manual/html_node/Auxiliary-Vector.html`

[8] We configured VARAN to emit an error message when an unhandled system call is encountered, and have implemented system call handlers on demand.

worked well for a low system call rate, but at higher rates the event pump quickly became a bottleneck.

As a result, we have instead opted for a design based on the Disruptor pattern [42], which uses a shared ring buffer allowing concurrent access by multiple producers and consumers, eliminating the need to dispatch events among queues, and thus improving both performance and memory consumption. Our implementation uses C11 atomics, in combination with cache aligning to achieve maximum performance with minimal use of locking (locks are used only during memory allocation and deallocation).

The size of the ring VARAN uses is configurable and has a default value of 256 events. Each event has a fixed size of 64 bytes; the size has been deliberately chosen to fit into a single cache line on modern x86 CPUs. This is sufficient for sending signals and system calls for which all arguments are passed by value (on x86-64, a system call can have up to six arguments of eight bytes, to fit into general purpose registers). However, for system call arguments passed by reference, the payload might have variable size and can be potentially larger than the event itself. In this case, we use events only to transfer shared pointers, which identify memory shared across versions.

The use of a shared memory buffer may result in a waste of system resources when the leader process performs a system call which blocks for a long period of time, as the followers use busy waiting to check for new events. To address this problem, we have introduced the concept of a *waitlock*. Whenever a follower makes a blocking system call, it acquires the waitlock. If there is no event available, the thread will block until the leader wakes up and notifies it. The waitlocks are efficiently implemented using a combination of C11 atomics and futexes [15].

3.3.2 Transferring file descriptors and leader replacement

Apart from the ring buffer, each version has a *data channel*, implemented using UNIX domain sockets. The data channel is used to send information which cannot be transferred via shared memory, in particular open file descriptors. Whenever the leader obtains a new file descriptor (*e.g.*, by opening a file), it sends this descriptor to all followers, effectively duplicating the descriptor into their processes. This is a crucial mechanism which enables the leader to be replaced transparently when it crashes. When the leader crashes, the follower that is elected as the new leader can simply continue executing using existing descriptors (*e.g.*, responding to requests coming over the network) without any disruption of service.

3.3.3 Multi-process and multi-threaded applications

Handling processes and threads is crucial in supporting many modern applications. In our design, we have opted to have separate ring buffers for each tuple of processes or threads in the system: for instance, when a process forks, the parent processes in the leader and all followers form one tuple, and

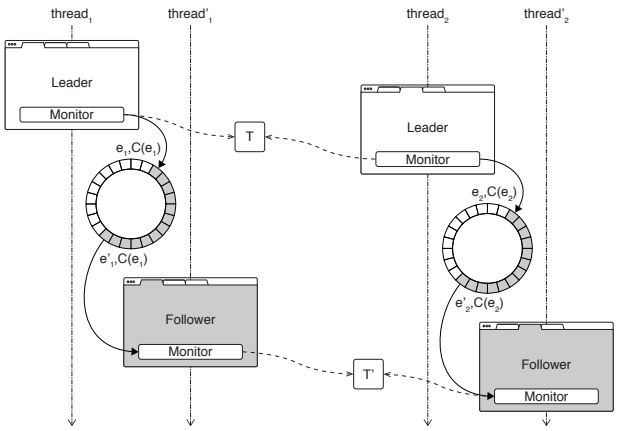

Figure 3. Event delivery in a multi-threaded NVX program, with the ordering of events enforced using logical clocks.

the child processes another, with a process in each tuple acting as the leader. More exactly, when a new process is forked, a new socket pair is established between the process and the coordinator and a new ring buffer is allocated. The leader then continues execution, but the coordinator waits until all followers fork a new process, establishing appropriate socket pairs for communication, and setting the child processes to read events from the newly-allocated ring buffer.

To alleviate non-determinism issues due to scheduling, VARAN enforces system call ordering across all tuples using Lamport's *happens-before* relation [28]. Currently, this is only implemented for multi-threaded applications, which make intensive use of synchronisation primitives, but the same solution could be employed for multi-process applications too.

Each variant has an internal Lamport clock, shared by all threads, and each event e_i sent through the ring buffer is annotated with a timestamp $C(e_i)$. Then, when replaying events from the buffer, each thread checks the timestamp of every new event and only receives the event if it does not violate the happens-before relation. This scenario is depicted in Figure 3. If $e_1 \rightarrow e_2$ (e_1 happens before e_2), then $C(e_1) < C(e_2)$ and VARAN enforces $e'_1 \rightarrow e'_2$. Without the ordering, there could be a situation where $e_1 \rightarrow e_2$, but $e'_1 \not\rightarrow e'_2$, which could lead to a divergence. A similar approach has been proposed in the past for record-replay in shared-memory systems [30].

To implement the internal clocks shared by the threads of a variant (T and T' in Figure 3), we use an atomic counter allocated in the shared memory space and updated using C11 atomics for efficiency. When the leader thread writes a new event into the ring buffer, it increments its variant's clock value and attaches it to the event. When a follower thread reads an event from the ring buffer, it compares its variant's clock value with the event's timestamp. If they are equal, the thread increments its variant's clock value and

processes the event, otherwise it continues waiting. Our current implementation uses busy waiting, as the wait times are expected to be small. However, shall this become a problem in the future, it is possible to use blocking wait instead (*e.g.*, a futex).

Our solution resembles existing deterministic multi-threading (DMT) mechanisms [4, 31]. The guarantees provided by VARAN are weaker than those typically provided by these systems as we do not enforce ordering across atomics-based synchronisation primitives. We have not detected any system call divergences caused by related data races in our benchmarks, which include multi-threaded applications (*e.g.*, *Redis*), similar to the experience reported for prior NVX systems. However, shall this become a problem, we could address it by employing a stronger form of determinism similar to existing DMT systems.

3.3.4 Memory allocation scheme

Efficient shared memory allocation plays an important role in a system like VARAN. We use a custom shared memory pool allocator implementation. The allocator has the notion of buckets for different allocation sizes, where each bucket holds a list of segments, and each segment is divided into chunks of the same size; each bucket holds a free list of chunks. When there are no more unused chunks in a bucket, the allocator requests a new segment from the memory pool, and divides it into chunks which are then added to the free list. Each bucket also has a lock associated with it which has to be held prior to an allocation from that bucket.

3.4 Rewrite rules for system call sequences

VARAN uses Berkeley Packet Filters (BPF) [34] to implement the system call rewrite rules introduced in Section 2.3. BPF is a machine language for writing rules and an interpreter shipped with many UNIX implementations, including Linux and BSD. BPF filters have been traditionally used to filter network packets, but recently also for system call filtering as a part of seccomp "mode 2" (also known as seccomp-bpf).

We have integrated a BPF interpreter in VARAN to allow for system call rewrite rules. Our implementation is based on the Linux kernel code which was ported to user space and extended for NVX execution. VARAN provides BPF extensions on top of the instruction set used by seccomp-bpf.[9] The `event` extension allows access to the event stream, which can be used to compare the system calls executed across versions, as we will show in Section 5.2.

The use of BPF has a number of advantages. First, it does not require the user to modify and recompile the monitor on every rule change. This is particularly important as rewrite rules can be application specific. Second, the BPF machine language was designed to be simple enough to prevent certain classes of errors—in particular, all filters are statically verified when loaded to ensure termination.

[9] `https://www.kernel.org/doc/Documentation/networking/filter.txt`

4. Performance evaluation

One of the main contributions of VARAN is a significantly lower performance overhead compared to existing state-of-the-art NVX systems. Therefore, we have conducted an extensive performance evaluation, using microbenchmarks (§4.1), high-performance C10k servers (§4.2) and applications used to evaluate prior NVX systems (§4.3).

The microbenchmarks were run on a four-core/eight-thread machine with a 3.50 GHz Intel Xeon E3-1280 CPU and 16 GB RAM running 64-bit Ubuntu 14.04 LTS, while the servers were run on a pair of such machines, one running the server under VARAN and the other the client. The machines are located in the same rack, connected by a 1 Gb Ethernet link.

4.1 Microbenchmarks

To measure the overhead introduced by VARAN while processing individual system calls, we designed a series of experiments that compare a system call intercepted and executed by VARAN against the same system call executed natively. We used five different system calls:

1. `close(-1)` is representative of an inexpensive system call, which returns immediately.
2. `write(DEV_NULL, ..., 512)` is representative of system calls which involve expensive I/O, but whose result can be sent entirely as a single event in the ring buffer.
3. `read(DEV_NULL, ..., 512)` is representative of system calls which involve expensive I/O, and whose result cannot be fully included in the associated event in the ring buffer. Instead, it has to be copied via additional shared memory (§3.3.1).
4. `open("/dev/null", O_RDONLY)` is representative of system calls that require transferring file descriptors (§3.3.2).
5. `time(NULL)` is a virtual system call implemented via the *vDSO* segment (§3.2.1). It internally calls `__vdso_time` (since glibc 2.15). We could not measure the overhead of using the *vsyscall* page, because it is deprecated on our system (and all recent versions of Linux), with all *vsyscalls* now redirected to their *syscall* versions.

We executed each system call one million times and computed the average of all execution times. Time measurements were done using the time stamp counter (*i.e.* the `RDTSC` instruction). Each set of measurements was preceded by a warm-up stage in which we executed the system call $10{,}000$ times.

Figure 4 shows the results. The first set of bars labeled *native* shows the execution time without VARAN. The second set of bars labeled *intercept* shows the execution time with interception, measuring the cost of binary rewriting: for

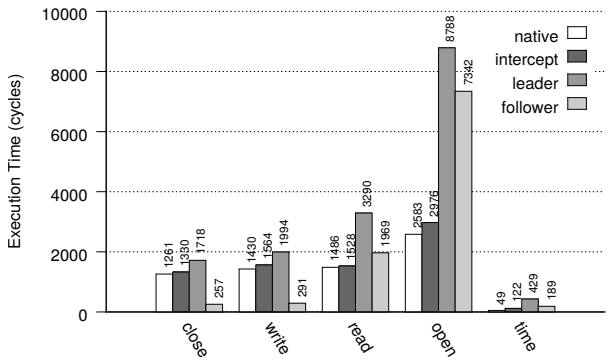

Figure 4. System call microbenchmarks.

these experiments, the intercepted system call is immediately executed, without any additional processing. As it can be seen, the interception cost is small, at under 15% in all cases except for `time`. The overhead of intercepting virtual system calls is high in relative terms, but low in absolute ones: 122 cycles vs 49 cycles for native execution for `time`.

The set of bars labeled *leader* shows the execution time for each system call to be intercepted, executed and recorded by the leader. That is, it is the sum of the *intercept* cost and the cost of recording the system call. For `close` and `write`, the overhead is only 36% and 39% respectively on top of native execution, because the arguments and results of these system calls can be recorded in a single event. For `read`, it is more expensive, at 139%, because transferring the result also involves accessing additional shared memory. Finally, the cost for `open` is the highest, since it also involves the slower transfer of the returned file descriptor via a UNIX domain socket.

Finally, the set of bars labelled *follower* shows the execution time of the follower, which has to intercept each system call and read its results from the ring buffer and (if necessary) shared memory. As expected, the costs for `close` and `write` are low (and significantly lower than executing the system call), because the entire result fits into a single event on the ring buffer. The costs for `read` and `open` are higher, because they involve additional shared memory and transferring a file descriptor, respectively, but they are still lower than the costs incurred by the leader.

4.2 C10k servers

Existing NVX systems, including those based on `ptrace`, can already run many (two-version) CPU-bound applications efficiently with an overhead typically less than 20%. As a result, we focus our evaluation on high-performance, heavily I/O-bound C10k servers which (1) represent the worst-case scenario for a system call monitor; and (2) form the backbone of modern, highly-scalable web applications, for which reliability is critical.

The five server applications used in our evaluations are summarized in Table 1 (the size is measured in lines of code, as reported by the *cloc* tool). For our performance experiments, we ran multiple instances of the same version of each application. Each experiment was performed six times, with the first measurement used to warm up the caches and discarded. The overhead was calculated as the median of the remaining five measurements.

Application	Size	Threading
Beanstalkd	6365	single-threaded
Lighttpd	38,590	single-threaded
Memcached	9779	multi-threaded
Nginx	101,852	multi-process
Redis	34,625	multi-threaded

Table 1. Server applications used in the evaluation.

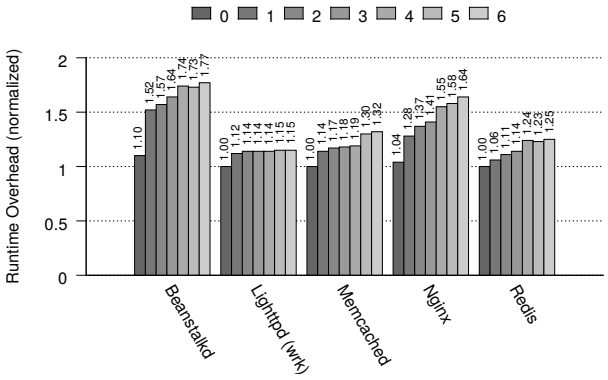

Figure 5. Performance overhead for the *Beanstalkd*, *Lighttpd*, *Memcached*, *Nginx* and *Redis* servers for different number of followers. The client and server are located on the same rack, simulating a worst-case scenario.

We give a short overview of each benchmark and the way in which we measure performance (namely throughput) in our experiments:

Beanstalkd is a simple and fast work queue, used by a number of websites to distribute jobs among workers. We used revision `157d88b` from the official *Git* repository, the latest revision at the time of writing. To measure performance, we used `beanstalkd-benchmark` with 10 concurrent workers each performing 10,000 push operations using 256 B of data per operation.

Lighttpd is a lightweight web server optimized for high performance environments. The version used for the measurements was 1.4.36, the latest version in the 1.4.x series at the time of writing. We measured the performance of serving a 4 kB page using `wrk`, which was run for 10 s with 10 clients.

Memcached is a high-performance, distributed memory object caching system, used by many high-profile websites to alleviate database load. We used revision 1.4.17, the latest at the time of writing. To measure the performance overhead, we used the *memslap* benchmark, part of the *libMemcached*

library. We used the default workload, *i.e.* an initial load of 10,000 key pairs and 10,000 test executions.

Nginx is a highly popular reverse proxy server often used as an HTTP web server, load balancer or cache. We used version 1.5.12, the latest at the time of writing. We measured performance using the same workload as for *Lighttpd*.

Redis is a high-performance in-memory, key-value data store, used by many well-known services. We used version 2.9.11 in our experiments. To measure performance, we used *redis-benchmark*, distributed as part of Redis. The benchmark issues different types of commands supported by Redis and measures both the throughput and the latency for each type. We used the default workload, *i.e.* 50 clients issuing 10,000 requests and calculated the average overhead across all commands.

Figure 5 shows the results for all servers. All performance numbers are obtained using the client-side tools mentioned above. Since the client machine is located on the same rack as the server, these numbers represent a worst-case scenario, as the network latency would hide some of the overhead for a more distant client machine.

For each benchmark, we show one bar, normalised relative to native execution, showing the performance of VARAN using a given number of followers. We stop at six followers, because our machine has eight threads, and we also need one thread for the leader and one for the coordinator.

The set of bars for 0 followers measure the interception overhead of VARAN using binary rewriting. This overhead is negligible for *Lighttpd*, *Memcached* and most *Redis* operations, 4% for *Nginx*, and 10% for *Beanstalkd*.

For all benchmarks, we see that the performance overhead increases slightly with the number of followers. For instance, the overhead for *Beanstalkd* increases from 1.52× for one follower to 1.77× for six followers, while the overhead for *Lighttpd* increases from 1.12× to 1.15×.

The figure also shows that there is a significant difference across benchmarks: the worst performer is *Beanstalkd*, which sees performance degradations in the range of 52% to 77%, while the best performers are *Lighttpd*, with only 12% to 15% overhead and some operations in *Redis* (not shown separately in Figure 5) with under 3% overhead.

4.3 Comparison with prior NVX systems

While Sections 4.1 and 4.2 illustrate the worst-case synthetic and real-world scenarios for a system call monitor, in order to compare VARAN directly with prior NVX systems, we have also run it on the same set of benchmarks used to evaluate prior systems. In particular, we chose to compare against three state-of-the-art NVX systems: *Mx* [21], *Orchestra* [40], and *Tachyon* [33]. These systems and their benchmarks are briefly described in the first three columns of Table 2. To our knowledge, we are the first to perform an extensive performance comparison of existing NVX systems.

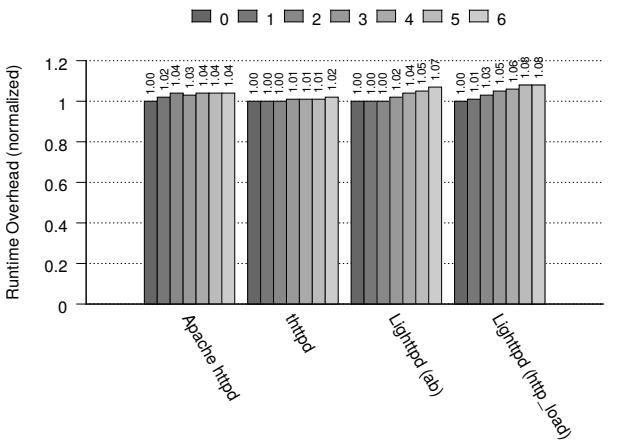

Figure 6. Performance overhead for the *Apache httpd*, *thttpd*, and *Lighttpd* servers for different numbers of followers to allow for comparison with existing systems.

The last two columns of Table 2 show the cumulative results. Since prior systems only handle two versions, the comparison is done against VARAN configured in the same way. However, we remind the reader that one of the strengths of VARAN's decentralised architecture is that it can often handle multiple versions with minimum additional overhead, and below we also show how VARAN performs on these benchmarks when more than two versions are used.

Apache httpd was used by *Orchestra*. We used version 1.3.29, the same as in the original work [40]. The overhead reported for *Orchestra* is 50% using the *ApacheBench* benchmark. VARAN introduces 2.4% overhead using the same benchmark, which is a significant improvement. Figure 6 shows the overhead introduced by VARAN for *Apache httpd* (and the other servers used to evaluate prior work) with different numbers of followers. As it can be seen, VARAN scales very well with increasing numbers of followers for these benchmarks.

Lighttpd has been used to evaluate both *Mx* and *Tachyon*. We used version 1.4.36. *Mx* used the *http_load* benchmark and reported 3.49× overhead while *Tachyon* used the *ApacheBench* benchmark and reported a 3.72× overhead. When benchmarked using *http_load*, VARAN introduced only 1.01× overhead, while with *ApacheBench* it introduced no noticeable overhead. In both cases, this marks a significant improvement over previous work.

thttpd was shown to introduce 1.17× overhead when run on top of *Tachyon* using the *ApacheBench* benchmark. When run on top of VARAN using the same settings as in [33], we have not measured any noticeable overhead.

Redis 1.3.8 was used in the evaluation of *Mx*. The performance overhead reported by *Mx* was 16.72× using the `redis-benchmark` utility. When run with VARAN using the same benchmark and the same workload, the overhead we

System	Mechanism	Benchmarks	Overhead	VARAN
Mx [21]	ptrace	*Lighttpd* (http_load)	3.49×	1.01×
		Redis (redis-benchmark)	16.72×	1.06×
		SPEC CPU2006	17.9%	14.2%
Orchestra [40]	ptrace	*Apache httpd* (ApacheBench)	50%	2.4%
		SPEC CPU2000	17%	11.3%
Tachyon [33]	ptrace	*Lighttpd* (ApacheBench)	3.72×	1.00×
		thttpd (ApacheBench)	1.17×	1.00×

Table 2. Comparison with *Mx*, *Orchestra* and *Tachyon* on the benchmarks used to evaluate these systems.

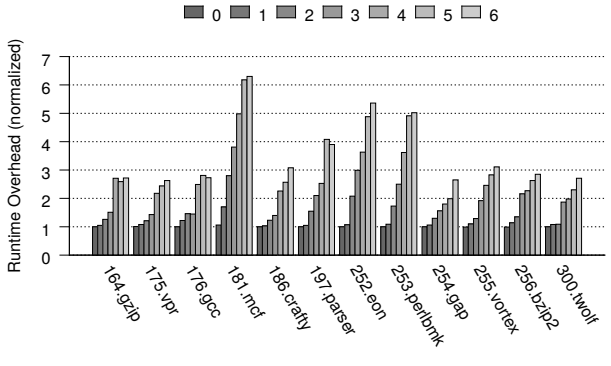

Figure 7. *SPEC CPU2000* performance overhead for different numbers of followers.

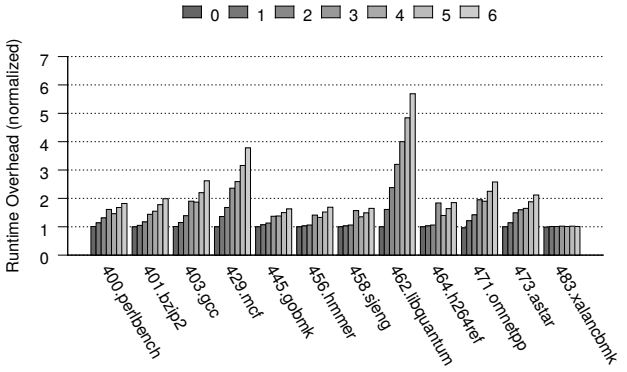

Figure 8. *SPEC CPU2006* performance overhead for different numbers of followers.

measured was 1.06×, which is again a significant improvement over previous work.

SPEC CPU2000 was used to evaluate *Orchestra*. We used the latest available version 1.3.1. *Orchestra* reported a 17% overhead, while VARAN introduced only a 11.3% overhead. The results for the individual applications contained in the *SPEC CPU2000* suite and for different numbers of followers can be seen in Figure 7. The reason these applications scale poorly with the number of followers is likely due to memory pressure and caching effects [24], and to the fact that our machine has only four physical cores (with two logical cores each). We plan to investigate these results in more detail in future work.

SPEC CPU2006 was previously used to evaluate *Mx*. We used the latest version 1.2. The overhead reported by *Mx* was 17.9%, while VARAN introduced only a 14.2% overhead. Individual results can be seen in Figure 8.

5. Application scenarios

VARAN is designed as a flexible framework that can support a variety of application scenarios involving NVX systems. In this section, we discuss four such scenarios: transparent failover (§5.1), multi-revision execution (§5.2), live sanitization (§5.3) and record-replay (§5.4).

5.1 Transparent Failover

NVX systems introduce a variety of opportunities for increasing software reliability and availability via transparent failover. For instance, one can run in parallel multiple variants of an application with different memory layouts, different software revisions or different implementations of a given interface to survive bugs that occur only in some of them.

VARAN makes it easy to implement transparent failover. When one of the versions crashes, the `SIGSEGV` signal handler installed in each version notifies the coordinator, which decides what restart strategy to use. When one of the followers crashes, the coordinator unsubscribes it from the list of currently-running followers, and discards it without affecting other followers. When the leader crashes, it designates one of the followers as the new leader (currently the one with the smallest internal ID), and notifies it to switch its system call table (§3.2) to that of the leader, and to restart the last system call while discarding the old (crashed) leader.

To demonstrate support for transparent failover, we reproduced a *Redis* bug[10] which was also used in the evaluation of *Mx* [21]. We ran in parallel eight consecutive revisions of *Redis* from the range `9a22de8` to `7fb16ba`, where the last revision introduced a bug which crashes the server by causing a segmentation fault. We then set up a client to send an `HMGET`

[10] `https://code.google.com/p/redis/issues/detail?id=344`

command that triggers the bug, and measured the increase in latency for that command. When the buggy version is a follower, we do not observe any increase in latency, as expected. When the buggy version is the leader, the latency increases from 42.36 µs to 122.62 µs. In both cases, we observed no extra degradation in throughput for the commands that follow.

As an additional experiment, we ran revisions `2437` and `2438` of *Lighttpd* (also used in the evaluation of *Mx*), the latter of which introduced a crash bug. We then set up a client that triggers the bug and measured the latency for that request. Both when the buggy version was the leader or a follower, there was no significant increase in latency, which remained at around 5 ms.

5.2 Multi-revision execution

Different software versions (revisions) can be run inside an NVX system as long as they all issue the same sequence of system calls [21]. This limitation is due to the fact that prior NVX systems run versions in lockstep (§2.3).

Because VARAN does not run the versions in lockstep and can use system call rewrite rules, it can often overcome this limitation. To illustrate, we used several *Lighttpd* revisions from the *Mx* [21] feasibility study which introduced new system calls and as such cannot be run in parallel by prior NVX systems that rely on lockstep execution.

As a first experiment, we ran revision `2435` as leader together with revision `2436` as follower. Revision `2436` introduces two additional checks using the `getuid` and `getgid` system calls. More precisely, revisions until and including `2435` used `geteuid()` and `getegid()` C library functions to check the user account under which the server is being run, before issuing an `open` system call. This resulted in a sequence of `geteuid`, `getegid` and `open` system calls. Revision `2436` replaced the use of the aformentioned functions with `issetugid()` which changed the system call sequence to `geteuid`, `getuid`, `getegid`, `getgid`, followed by `open` as before.

To allow for this divergence, we used the custom BPF filter shown in Listing 1. The filter is executed by the follower whenever a divergence is detected. In our experiment, this happens when the follower executes the newly introduced `getuid` system call. The filter first loads the system call number executed by the leader into the implicit BPF accumulator (line 1) and checks whether the call is either `getegid` (line 2) or `open` (line 3). The former will be true in this case, so control will transfer to line 6, which loads the system call number executed by the follower into the accumulator, checks whether it is `getuid` (line 7) and finally transfers control to line 12 returning the value `SECCOMP_RET_ALLOW`, which instructs VARAN to execute the additional system call (*i.e.* `getuid`) in the follower. Any other combination of system calls would have killed the follower (line 11). After executing the `getuid` system call and replaying the execution of `getegid` (which the leader also executed), VARAN would detect a second divergence when the follower tries to execute `getgid` instead of `open`. This divergence would be resolved using the same filter, taking the path on lines 3, 9, 10 and 12.

```
1  ld event[0]
2  jeq #108, getegid /* __NR_getegid */
3  jeq #2, open /* __NR_open */
4  jmp bad
5  getegid:
6  ld [0] /* offsetof(struct seccomp_data, nr) */
7  jeq #102, good /* __NR_getuid */
8  open:
9  ld [0] /* offsetof(struct seccomp_data, nr) */
10 jeq #104, good /* __NR_getgid */
11 bad: ret #0 /* SECCOMP_RET_KILL */
12 good: ret #0x7fff0000 /* SECCOMP_RET_ALLOW */
```

Listing 1. Example of a BPF rewriting rule.

Note this is only one possible filter for allowing this divergence; in particular, one could write a filter that takes into account more information about the context in which it should be applied, *e.g.*, by inspecting some system call arguments.

We used a similar filter to run revisions `2523` and `2524`, the latter of which introduces an additional `read` system call to access the `/dev/urandom` file to obtain an additional source of entropy. We were also able to run revisions `2577` and `2578` where the difference consists of an additional `fcntl` system call to set a `FD_CLOEXEC` flag on one of the file descriptors.

Currently, VARAN's implementation can use BPF filters only to allow adding or removing system calls in followers. However, this is not a fundamental limitation, and in the future we plan to support other types of transformations, such as replacing one sequence of system calls with another.

5.3 Live Sanitization

Sanitization is one of the most effective testing techniques for revealing low-level bugs such as uninitialised pointer dereferences and use-after-free errors. Both Clang and GNU C Compiler now include a set of sanitizers—AddressSanitizer (ASan), MemorySanitizer (MSan), ThreadSanitizer (TSan)—which can be used to statically instrument the code with various checks. Unfortunately, these checks introduce extra overhead (*e.g.*, 2× for ASan, 3× for MSan and 5-15× for TSan). which is why these sanitizers are typically only used in offline testing. However, during testing developers only use a limited set of inputs which might not reveal all bugs.

One possible solution is to record execution traces during deployment and then replay them in a testing environment with sanitization enabled. However, this approach is unlikely to work in practice for several reasons. First, since we do not know in advance which traces are potentially interesting (*e.g.*, trigger sanitization checks) and which are not, we have to potentially collect and replay a huge number of execution traces. Even with some form of deduplication, this is usually impractical. Second, for long-running applications such as servers, the log will quickly grow to a large size. Third, many

customers will refuse to share the logs from their production deployment.

With VARAN, we can perform live sanitization by running the native unsanitized version as the leader, with sanitized versions as followers. While sanitization itself introduces a performance overhead, since followers do not need to execute any I/O operations and merely replay them, they can often keep up with the leader, allowing users to run sanitized versions in production without introducing any significant overhead. Note that VARAN's architecture also provides the ability to run several sanitizers concurrently, which is important because many sanitizers are mutually incompatible.

To demonstrate this, we build revision `7f77235` of *Redis* twice: once with Clang without any sanitization, once with ASan enabled. We then ran both versions in parallel using VARAN and used the same benchmark with the same settings as for our performance evaluation (§4.2). As expected, we have not measured any additional slowdown in the leader compared to the scenario with two non-sanitized versions being run in parallel. To get a better insight into the effect of running the sanitized version with VARAN, we have also measured the median size of the log, *i.e.* the distance between the leader and the follower. This value is only six events, which does not impose any problems.

5.4 Record-Replay

Although VARAN shares similarities with record-replay systems, there are significant differences; in particular, the log is of fixed size and only kept in-memory. However, it is possible to easily extend VARAN to provide full record-replay capabilities by implementing two artificial clients: (i) during the record phase, one acting as a follower whose only goal is to write the content of the ring buffer to persistent storage, and (ii) during the replay phase, one acting as the leader, reading the content of the log from the persistent storage and publishing events into the ring buffer for consumption by replay clients.

Compared to some of the previous record-replay systems, VARAN has a number of advantages. First, decoupling the logic responsible for reading/writing the log from the actual application into a separate process allows the application to run at nearly full speed and utilise the multiple cores available in modern CPUs. Second, since VARAN was designed to run multiple instances at the same time, we can replay multiple versions at once, *e.g.*, to determine which versions of the application from a given range are susceptible to a crash reported by the user.

We have implemented a simple prototype of the two aforementioned clients on top of VARAN and compared its performance against *Scribe* [27], a state-of-the-art record-replay system implemented in the kernel. Unfortunately, because *Scribe* is implemented in the kernel and is only maintained for an old 32-bit Linux kernel (2.6.35), we had to run our experiments inside a virtual machine (kindly provided to us by *Scribe*'s authors, as the source tree was broken at the time of our experiments). To allow for a more faithful comparison, we ran VARAN inside the same virtual machine.

We used *Redis* as a benchmark, running the same workload as before, and configured both systems to record the execution to persistent storage. We recorded an overhead of 53% for *Scribe*,[11] compared to 14% for VARAN.

6. Discussion

This section discusses some of the implications of VARAN's design, including its main limitations, many of which are inherent to all existing NVX systems.

CPU utilisation and memory consumption. The performance evaluation reported in Section 4 considers the overhead in terms of throughput or clock time. However, an NVX framework introduces a CPU utilisation overhead linear in the number of versions. While this might be a serious concern in some scenarios, leaving cores idle has a cost as well [3] and in many cases idle cores can be profitably used to increase software reliability and security [5, 8, 9, 13, 40].

Similarly, the memory overhead imposed by VARAN is linear in the number of versions, as in prior NVX systems. This can lead to degradations in performance due to memory pressure and caching effects, as we have observed in Section 4.3.

Memory-based communication. As prior NVX systems, VARAN does not support memory-based communication. More exactly, VARAN only allows files to be mapped into memory as read-only—if the file would be mapped as read-write, any writes by the leader would likely lead to divergences in followers, as they would read the value written by the leader rather than the original value. This limitation comes from the fact that memory-based communication cannot be intercepted by interposing upon the system call interface, and as such is invisible to NVX systems operating at the system call level.

Synchronisation. While VARAN supports multi-threaded and multi-process applications (§3.3.3), there is a potential issue with synchronisation primitives implemented entirely in user space, as these primitives will be invisible to VARAN. While it is possible to use entirely user-space synchronisation primitives, in our experience, they are not that frequent and standard synchronisation primitives combine atomics with system calls (*i.e.* futex). We have not observed any related problems in our concurrent benchmarks (§4.2, §4.3).

Security. Although our focus with VARAN has been on improving software reliability, VARAN could be also used to implement existing NVX security defences [13, 40]. However, there are two additional problems that VARAN introduces, as discussed below.

[11] The overhead we measured for *Scribe* is higher than that reported in [27]; however, note that the original work used less I/O intensive benchmarks such as *Apache httpd* and that the use of a virtual machine also affected the result.

First, the use of buffering, while essential for improving performance, leads to delayed detection of divergences, providing attackers with a window of opportunity in which to perform malicious system calls. However, VARAN's buffer size is configurable, and could be set to one to disable buffering. Even without buffering, VARAN's binary rewriting mechanism is more efficient than `ptrace`-based solutions.

Second, since VARAN resides in the same address space as the application, a return-oriented programming (ROP) attack can bypass VARAN's tracing mechanism and thus escape detection. Furthermore, VARAN's code could be a primary target of such an attack. However, this is partially mitigated by the fact that VARAN's code is loaded at a random memory address.

7. Related Work

N-version programming was introduced in the seventies by Chen and Avizienis [11]. The core idea was to have multiple teams of programmers develop the same software independently and then run the produced implementations in parallel in order to improve the fault tolerance of the overall system. Both version generation and the synchronisation mechanism required manual effort.

Recent work on NVX systems has moved in the direction of opportunistically using existing versions—*e.g.*, different browser implementations [47] or different software revisions [21]—or automatically synthesising them—*e.g.*, by varying the memory layout [5] or the direction of stack growth [40]. Significant effort has also been expended on running multiple versions in parallel in the context of online and offline testing [33, 41, 43, 45]. VARAN targets NVX systems that use system call level synchronisation and is oblivious to the way in which the versions are generated.

Recent NVX systems that synchronise versions at the level of system calls use either the `ptrace` interface [21, 33, 40] or kernel modifications [13] to implement monitors. As discussed, `ptrace`-based systems incur an unacceptable overhead on I/O-bound applications—for instance, *Tachyon* [33] reports an overhead of 3.72× on *Lighttpd* and *Mx* [21] an overhead of 16.72× in one of the Redis experiments. By contrast, kernel-based systems [13] achieve overheads competitive to VARAN—but the main disadvantages are the additional privileges required for deployment and the difficulty of maintaining and debugging the kernel patches. Finally, all existing NVX systems operating at the level of system calls, both user- and kernel-level, require lockstep execution, which introduces significant limitations both in terms of performance and flexibility, as discussed in detail in Sections 2.2 and 3.4.

Event streaming in VARAN can be seen as a variant of record-replay. However, unlike traditional record-replay systems that require a persistent log [17, 20, 38, 39], VARAN keeps the shared ring buffer in memory, and deallocates events as soon as they are not needed, which minimises performance overhead and space requirements in the NVX context. As we showed in Section 5.4, VARAN can also be efficiently extended into a traditional record-replay framework.

RR [6] replicates an application into multiple instances for fault tolerance. It also uses a variant of record-replay to synchronise the multiple instances, but in contrast to VARAN RR is focused on fail-stop scenarios involving identical replicas. RR was implemented as a Linux kernel extension, which as discussed above presents several disadvantages compared to a user-level solution like VARAN, and was evaluated on a single benchmark.

More generally, system call interposition has been an active area of research [1, 2, 19, 23, 25, 36, 46]. VARAN draws inspiration from the Ostia delegating architecture [16], and from the selective binary rewriting approach implemented by *BIRD* [35] and *seccompsandbox*[12] (our binary rewriting implementation being based on the latter).

8. Conclusion

Recent years have seen a growing interest in using NVX systems as a way to increase the reliability and security of software systems. While NVX systems hold promise, frameworks for implementing them efficiently have lagged behind.

In this paper, we have introduced VARAN, a novel architecture for implementing NVX monitors. VARAN combines selective binary rewriting with high-performance event streaming to deliver a flexible and efficient user-space solution that incurs a low performance overhead, can scale to large numbers of versions, is easier to debug than prior systems, and can handle small divergences in the sequences of system calls issued across versions.

Our experimental evaluation has demonstrated that VARAN can run C10k network servers with low performance overhead and can be used in various scenarios such as transparent failover, multi-revision execution, live sanitization and record-replay.

For up-to-date information about the project, please visit `http://srg.doc.ic.ac.uk/projects/varan`.

Acknowledgments

We would like to thank Oscar Dustmann, Paul Marinescu, Luis Pina, Bennet Yee and the anonymous reviewers for their valuable comments on the paper. Special thanks to our ASPLOS shepherd, Andrew Baumann, for his time and constructive feedback. Petr is a recipient of a Google Europe Fellowship in Software Engineering, and Cristian of an Early-Career EPSRC Fellowship, and this research has been generously supported by these fellowships.

[12] `https://code.google.com/p/seccompsandbox/`

References

[1] A. Acharya and M. Raje. MAPbox: Using parameterized behavior classes to confine applications. In *Proc. of the 9th USENIX Security Symposium (USENIX Security'00)*, Aug. 2000.

[2] A. Alexandrov, P. Kmiec, and K. Schauser. Consh: Confined execution environment for Internet computations. `http://itslab.inf.kyushu-u.ac.jp/ssr/Links/alexandrov98consh.pdf`, Dec. 1998.

[3] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40:33–37, 2007.

[4] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *Proc. of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*, Mar. 2010.

[5] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'06)*, June 2006.

[6] P. Bergheaud, D. Subhraveti, and M. Vertes. Fault tolerance in multiprocessor systems via application cloning. In *Proc. of the 27th IEEE International Conference on Distributed Computing Systems (ICDCS'07)*, June 2007.

[7] D. Bruschi, L. Cavallaro, and A. Lanzi. Diversified process replicae for defeating memory error exploits. In *Proc. of the International Performance, Computing, and Communications Conference (IPCCC'07)*, Apr. 2007.

[8] C. Cadar and P. Hosek. Multi-version software updates. In *Proc. of the 4th Workshop on Hot Topics in Software Upgrades (HotSWUp'12)*, June 2012.

[9] C. Cadar, P. Pietzuch, and A. L. Wolf. Multiplicity computing: A vision of software engineering for next-generation computing platform applications. In *Proc. of the FSE/SDP workshop on the Future of Software Engineering Research (FoSER'10)*, Nov. 2010.

[10] R. Capizzi, A. Long, V. Venkatakrishnan, and A. P. Sistla. Preventing information leaks through shadow executions. In *Proc. of the 24th Annual Computer Security Applications Conference (ACSAC'08)*, Dec. 2008.

[11] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. of the 8th IEEE International Symposium on Fault Tolerant Computing (FTCS'78)*, June 1978.

[12] Chromium.org. Linux Zygote: The use of zygotes on Linux. `https://code.google.com/p/chromium/wiki/LinuxZygote`.

[13] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: a secretless framework for security through diversity. In *Proc. of the 15th USENIX Security Symposium (USENIX Security'06)*, July-Aug. 2006.

[14] D. Devries and F. Piessens. Noninterference through secure multi-execution. In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'10)*, May 2010.

[15] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Proc. of the 2002 Ottawa Linux Symposium (OLS'02)*, June 2002.

[16] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proc. of the 11th Network and Distributed System Security Symposium (NDSS'04)*, Feb. 2004.

[17] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *Proc. of the 2006 USENIX Annual Technical Conference (USENIX ATC'06)*, May-June 2006.

[18] J. Giacomoni, T. Moseley, and M. Vachharajani. FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *Proc. of the 13th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, Feb. 2008.

[19] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *Proc. of the 6th USENIX Security Symposium (USENIX Security'96)*, July 1996.

[20] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, Dec. 2008.

[21] P. Hosek and C. Cadar. Safe software updates via multi-version execution. In *Proc. of the 35th International Conference on Software Engineering (ICSE'13)*, May 2013.

[22] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proc. of the 3rd USENIX Windows NT Symposium (USENIX NT'99)*, July 1999.

[23] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Proc. of the 6th Network and Distributed System Security Symposium (NDSS'99)*, Feb. 1999.

[24] A. Jaleel. Memory characterization of workloads using instrumentation-driven simulation. Technical report, Intel Corporation, 2007.

[25] T. Kim and N. Zeldovich. Practical and effective sandboxing for non-root users. In *Proc. of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*, June 2013.

[26] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proc. of the 11th USENIX Security Symposium (USENIX Security'02)*, Aug. 2002.

[27] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proc. of the ACM SIGMETRICS 2010 (SIGMETRICS'10)*, June 2010.

[28] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the Association for Computing Machinery (CACM)*, 21(7):558–565, July 1978. ISSN 0001-0782.

[29] P. P.-C. Lee, T. Bu, and G. Chandranmenon. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In *Proc. of the 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS'10)*, Apr. 2010.

[30] L. Levrouw, K. Audenaert, and J. Van Campenhout. A new trace and replay system for shared memory programs based on Lamport clocks. In *Proc. of the 6th IEEE International Parallel & Distributed Processing Symposium (IPDPS'94)*, Oct. 1994.

[31] T. Liu, C. Curtsinger, and E. D. Berger. DTHREADS: Efficient deterministic multithreading. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, Oct. 2011.

[32] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'05)*, June 2005.

[33] M. Maurer and D. Brumley. TACHYON: Tandem execution for efficient live patch testing. In *Proc. of the 21st USENIX Security Symposium (USENIX Security'12)*, Aug. 2012.

[34] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. of the 1993 Winter USENIX Conference*, Jan. 1993.

[35] S. Nanda, W. Li, L.-C. Lam, and T. cker Chiueh. BIRD: Binary interpretation using runtime disassembly. In *Proc. of the 4th International Symposium on Code Generation and Optimization (CGO'06)*, Mar. 2006.

[36] N. Provos. Improving host security with system call policies. In *Proc. of the 11th USENIX Security Symposium (USENIX Security'02)*, Aug. 2002.

[37] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2:1–2:34, Mar. 2012.

[38] M. Ronsse and K. De Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems (TOCS)*, 17(2):133–152, May 1999. ISSN 0734-2071.

[39] Y. Saito. Jockey: a user-space library for record-replay debugging. In *Proc. of the 6th International Workshop on Automated Debugging (AADEBUG'05)*, Sept. 2005.

[40] B. Salamat, T. Jackson, A. Gal, and M. Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proc. of the 4th European Conference on Computer Systems (EuroSys'09)*, Mar.-Apr. 2009.

[41] S. Sidiroglou, S. Ioannidis, and A. D. Keromytis. Band-aid patching. In *Proc. of the 3rd Workshop on Hot Topics in System Dependability (HotDep'07)*, June 2007.

[42] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. Technical report, LMAX, 2011. URL `http://lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf`.

[43] J. Tucek, W. Xiong, and Y. Zhou. Efficient online validation with delta execution. In *Proc. of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*, Mar. 2009.

[44] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, Oct. 2011.

[45] M. A. Vouk. Back-to-back testing. *Information and Software Technology (IST)*, 32:34–45, Jan.-Feb. 1990.

[46] D. A. Wagner. Janus: an approach for confinement of untrusted applications. Technical Report UCB/CSD-99-1056, University of California at Berkley, 1999. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/1999/5271.html`.

[47] H. Xue, N. Dautenhahn, and S. T. King. Using replicated execution for a more secure and reliable web browser. In *Proc. of the 19th Network and Distributed System Security Symposium (NDSS'12)*, Feb. 2012.

[48] A. R. Yumerefendi, B. Mickle, and L. P. Cox. Tightlip: Keeping applications from spilling the beans. In *Proc. of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI'07)*, Apr. 2007.