Taming Parallelism in a Multi-Variant Execution Environment

Stijn Volckaert
University of California, Irvine
stijnv@uci.edu

Bart Coppens
Ghent University
bart.coppens@ugent.be

Bjorn De Sutter
Ghent University
bjorn.desutter@ugent.be

Koen De Bosschere
Ghent University
koen.debosschere@ugent.be

Per Larsen
University of California, Irvine
perl@uci.edu

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

Abstract

Exploit mitigations, by themselves, do not stop determined and well-resourced adversaries from compromising vulnerable software through memory corruption. Multi-variant execution environments (MVEEs) add additional assurance by executing multiple, diversified copies (variants) of the same program in lockstep while monitoring their behavior for signs of attacks (divergence). While executing multiple copies of the same program requires additional computational resources, modern MVEEs run many workloads at near-native speed and can detect adversaries before they leak secrets or achieve persistence on the host system.

Multi-threaded programs are challenging to execute in lockstep by an MVEE. If the threads in a set of variants are not scheduled in the exact same order, the variants will diverge from each other in terms of the system calls they make. While benign, such divergence undermines the MVEEs ability detect divergence caused by malicious program inputs. To address this problem, we developed an MVEE-specific synchronization scheme that lets us execute a set of multi-threaded variants in lockstep without causing benign divergence. Our fully-fledged MVEE runs the PARSEC 2.1 and SPLASH-2x parallel benchmarks (with four worker threads per variant) with a slowdown of less than 15% relative to unprotected execution. Addressing this longstanding compatibility issue makes MVEEs a viable defense for a far greater range of realistic workloads.

1. Introduction

Modern compilers and operating systems employ a range of countermeasures that make it far harder to turn a memory corruption vulnerability into an exploit. Data execution prevention [33], for instance, has all but obsoleted the classic code injection techniques. However, other exploitation techniques such as code reuse, and non-control-data and information leakage attacks remain viable.

Much effort has been spent searching for new and improved mitigations. While proposals for stronger mitigations do increase resilience, they are narrowly tailored to target certain classes of attacks—typically variants of code reuse. Control-flow integrity [1], for example, increase resilience against code reuse over the current address space layout randomization techniques but is by no means impervious to sophisticated code reuse attacks that avoid any "illegitimate" control flows, and offers no protection of non-control data.

Multi-Variant Execution Environments (MVEEs) have become a hot research topic due to their potential to break the seemingly endless cycle of new mitigations being bypassed by new exploits which are then addressed by yet more mitigations [13, 19, 21, 30, 37, 44, 45]. The fundamental idea is to execute two or more functionally equivalent programs (variants) in lockstep and monitor their behavior at the level of system calls. MVEEs terminate execution upon detection of divergence. Because each diversified variant receives the same program input but responds differently to memory corruption, it becomes exceedingly hard to simultaneously and reliably exploit N program variants without causing them to diverge.

A major obstacle to broader adoption of MVEE techniques is multithreading. The order of system calls made by a variant depends on the order in which its threads are scheduled. In other words, if the thread schedules between two variants diverge, so will their externally visible behavior. This is a problem because MVEEs rely on divergence to detect malicious or unintended behavior. Moreover, by allowing "benign" divergence, the MVEE implicitly allows its variants to receive inconsistent inputs (i.e., one variant might receive inputs that the other variants have not). These inputs can propagate through the program execution, leading to yet

EuroSys '17 April 23–26, 2017, Belgrade, Serbia

 ISBN 978-1-4503-4938-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3064176.3064178

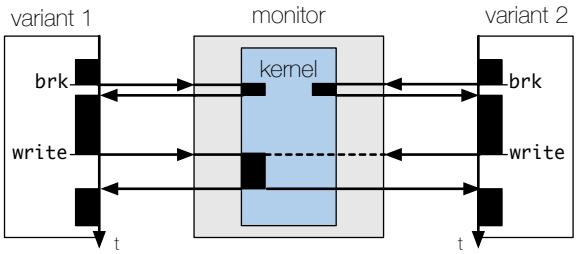


Figure 1. Monitoring and replication between two variants.

more divergent behavior, until a point where the MVEE can no longer tell benign divergence apart from "true" divergence caused by an attack. In summary, once two variants have diverged, they are unlikely to converge again.

This paper identifies the types of thread interactions that must be ordered to avoid benign divergence. We then present an MVEE-specific way to efficiently order the thread schedules within each variant to match those of the other variants. To the best of our knowledge, our work is the first in-depth exploration of techniques supporting execution of multi-threaded programs under MVEEs. In summary, we make the following contributions:

- We evaluate multiple strategies to capture the order of inter-thread interactions in one variant and replay them in the others. We use our findings to develop a better-performing synchronization agent. Our "wall-of-clocks" agent addresses thread scheduling and resource contention issues we identified in simpler synchronization strategies; this leads to substantially lower overheads.
- We extend ReMon and GHUMVEE, our own security-oriented MVEEs, to transparently inject the synchronization agent into the variants' address spaces. To the best of our knowledge, this extension makes our MVEEs the first to support multithreaded variants in which the threads interact explicitly, although limited developer assistance may be necessary to prepare the variants.
- We perform a careful and detailed evaluation showing that our wall-of-clocks synchronization agent is highly efficient and practical.

2. Background

All MVEEs run two or more variants side by side on the same machine and use a monitor to compare the variants' behavior at the level of system calls. The monitor must synchronize the variants and present them as a single application to the end-user. To do so, the monitor duplicates program inputs once for each variant; variant outputs are similarly compared and deduplicated such that each output operation is performed only once. MVEEs monitor variants by interposing on the system calls made by each variant. In other words, the monitor gains control over a variant each time it makes a system call. As shown for the handling of two system calls (brk and write) in Figure 1, this lets the monitor decide whether or not to forward the system call to the kernel, and if and when variant execution is resumed.

The variant synchronization model is a key differentiator among MVEEs. MVEEs that are used for the reliability purposes[1] use a relaxed synchronization model in which a designated leader variant may run ahead of the follower variants. Under this model, the results of the system calls made by the leader are stored in a shared *ring buffer* such that they can subsequently be copied to the followers. This reliability-oriented model can tolerate minor differences in the variants' behavior resulting from nondeterminism[2]. Security-oriented MVEEs use a strict synchronization model where variants execute each system call in lockstep; no variant is allowed to proceed past (certain) system calls until all variants have made an equivalent system call. Such MVEEs cannot tolerate behavioral differences that affect the ordering of system calls or the arguments of such calls.

We also distinguish MVEEs based on the monitor execution context. Early MVEEs executed a single, centralized monitor in a separate process or in the kernel. In both cases, the monitor executed in a different context than the variants [12, 13, 18, 30, 37, 43]. More recent work has explored decentralized monitors that run in the same execution context as the variants to some degree [19, 21, 45]. In the decentralized designs, the monitor can intercept system calls directly, whereas the external monitor designs require context switching to invoke the monitor. The downside of decentralized approaches is that coordination of multiple monitors inside the variants adds additional implementation complexity.

2.1 Handling Parallelism

Single-threaded variants produce the same output when given the same inputs[3]. This property does not extend to multithreaded variants in which the threads communicate; in this case, the output also depends on the thread interleaving at run time. To run multithreaded programs with strict variant synchronization (lockstepping), the MVEE must therefore constrain thread interleavings such that each variant thread makes system calls in the same order and with the same arguments as the equivalent threads in other variants. Failure to do this leads to "benign" divergent behavior, which, if the MVEE tolerates it, can lead to the variants receiving inconsistent program inputs. These inconsistent inputs can then propagate through the program's execution, where they can trigger even more divergence, until a point is potentially reached where the MVEE can no longer distinguish "benign" divergence from "true" divergence caused by attacks. To guarantee that this does not occur, the MVEE can simply not tolerate "benign" divergent behavior at all.

[1] For example: to run an older and newer release of the same program side by side to ensure that a patch preserves program functionality [19].

[2] Programs are nondeterministic if their behavior can change from run to run, even when receiving the same program inputs.

[3] Some exceptions apply. We refer interested readers to the literature for an in-depth discussion [37, 43].

We can address this challenge in one of two ways. First, we can use deterministic multithreading (DMT) to force variants to repeat the same thread interleaving (schedule) when given the same inputs [6, 9, 11, 14, 15, 28, 29, 31, 32, 34, 46]. Dividing the program execution into serial and parallel phases is a common way to establish this schedule. In a serial phase, all threads run concurrently but they may not modify the shared program state. In a parallel phase, threads may commit to the shared state, but only one thread may run at any given time. Threads may stay in a parallel phase until their allotted quantum ends. This quantum cannot be based on time, as background activity on the system may affect how quickly a thread progresses. Instead, DMT systems allocate quanta based on logical thread progress. Logical thread progression can be measured efficiently using hardware performance counters. More recent DMT systems use different scheduling algorithms but also rely on performance counters to measure logical thread progress.

Using performance counters to quantify progress is not ideal in context of MVEEs because the software diversity techniques applied to variants are likely to affect the number of instructions and other measures of progress [23]. Applying DMT to diversified program variants could lead to each of them having a fixed, but different schedule which does not eliminate the possibility of "benign divergence".

The second alternative is to accept nondeterminism and merely require that all variants execute in the same nondeterministic order. Online Record/Replay (R+R) systems can provide this guarantee by logging the execution in one variant and replaying it in the other variants [7, 8, 25]. R+R systems do not rely on measuring thread progress and are therefore less sensitive to variations in the program execution introduced by software diversification. Some challenges must be overcome to deploy an R+R in a security-oriented MVEE, however. We explain how we addressed these in Section 3.

3. Design

Our goal is to constrain the execution of variants such that they all use the same, possibly nondeterministic, thread interleaving. Our work targets data race-free programs. This is consistent with related work on weakly-deterministic DMT systems [14, 32], and Record+Replay [7, 35, 36]. Removing unintended data races is an important, but orthogonal problem. Two types of interactions can affect the variants' behavior: (i) system calls operating on shared resources, and (ii) inter-thread communication via shared memory.

3.1 System Calls

The threads in a multithreaded variant can influence each other's system call results even if these threads do not communicate directly. A trivial example of such cases is when multiple threads within the same process open files. The kernel assigns the first available file descriptor (FD) to each newly opened file. The order in which threads open files can therefore affect which FDs are returned to which thread. Consider for example a program in which two threads simultaneously open files and print out the FD values assigned to these files. If an MVEE runs two variants of this program in parallel, and the MVEE does not enforce an ordering on the `sys_open` calls used to open these files, then different FD values might be returned to the equivalent sets of threads in these variants. The MVEE will then detect divergence when the FD values are printed out, or when the FD values are used in subsequent file operations. This particular problem, as well as similar problems that can occur for other system calls, can be circumvented by forcing the MVEE to replicate the master variant's system call results to the slave variants. This would add additional complexity to the MVEE, however. If the results of one system call (e.g., `sys_open`) are used as the arguments of a later system call (e.g., `sys_mmap`), the MVEE would need to silently overwrite the arguments of the later call with the intended values. Thus, we generally opt for the simpler solution of forcing related system calls to execute in the same order in all variants.

3.2 Inter-Thread Communication

Ordering system calls suffices to eliminate divergence in loosely-coupled multithreaded variants (i.e., variants in which the threads do not communicate directly). Virtually every multithreaded program contains at least some degree of direct inter-thread communication, however. Forcing all communicating instructions (i.e., instructions that access memory shared between threads) to execute in the same order in all variants would trivially eliminate divergence, but is also prohibitively expensive [15]. Instead, we propose to force the variants to access synchronization variables (e.g., locks, condition variables, ...) in the same order. The underlying idea is that communicating instructions are, by definition, protected using synchronization primitives in data-race-free programs. Controlling access to synchronization variables is therefore equivalent to ordering individual instructions in such programs, but comes at a much smaller performance cost [14, 32].

Our design records the order in which the master (or "leader") variant accesses synchronization variables and imposes that same order in the slave variants. This is done by instrumenting synchronization operations or *sync ops*—we use these terms interchangeably. Related work typically considers functions (e.g., `pthread_mutex_lock`) that access synchronization variables to be sync ops. In this paper, however, we use the term sync op to refer to individual instructions accessing synchronization variables. The instrumentation code we add contains calls to a *synchronization agent*. This agent is implemented as a shared library that our MVEE injects into the address space of each variant.

The master variant's synchronization agent captures the order in which the master executes its sync ops, and records this order in the synchronization buffer or *sync buffer*, as shown in Figure 2. This buffer is a memory segment shared between all variants. The slave variants' synchronization agents ensure that sync ops execute in an order that is equivalent to the order logged in the sync buffer. To enforce this ordering, we temporarily suspend the execution of any slave

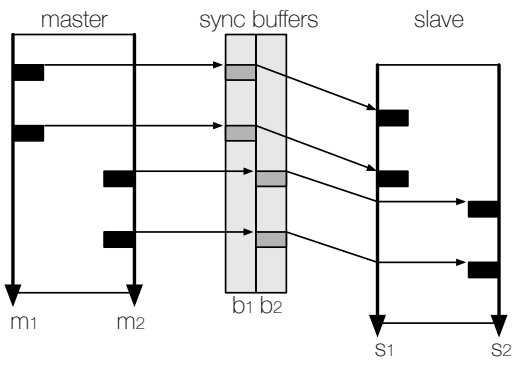


Figure 2. Synchronization through the shared sync buffer.

variant thread whenever it is about to execute a sync op that would violate the intended order. The execution of the thread is resumed when all of the sync ops that must precede the suspended sync op have completed.

3.3 Requirements and Challenges

Astute readers may notice that our system shares its goals with online R+R systems. The key difference is that it addresses challenges that are unique to MVEEs. Online R+R system enforce identical *input/output behavior* in a set of *identical* programs. MVEEs must enforce identical *system call behavior* in a set of *diverse* program variants.

I/O operations are only a subset of the system calls that must typically be executed in lockstep in an MVEE. Thus, online R+R systems must order only the sync ops that affect explicit I/O behavior. Such sync ops are often easy to identify as they implement known and well-understood synchronization primitives such as `pthread` mutexes, and are often contained within dedicated threading libraries (e.g., `libpthread` or `libgomp`). An MVEE must also order sync ops that affect non-I/O system calls. This can be significantly harder to do. The memory allocator in GNU's `libc`, for instance, protects its internal data structures using low-level synchronization primitives (e.g., assembly-based spinlocks). The sync ops that implement such primitives are often inlined and scattered throughout the program binaries. Failure to order these low-level sync ops may affect the program's behavior with respect to memory-related system calls such as `sys_mprotect`.

The fact that variants are diversified creates additional challenges. Code and data addresses will differ randomly among the program variants. Thus, the same logical synchronization variable (e.g., a mutex) might be located at a different address in each variant. MVEEs must maintain a mapping between such addresses in order to enforce equivalent synchronization operation orders.

Finally, the synchronization agent must ensure that it does not introduce any observable divergence. This is not trivial, as the agent performs different functions in the master variant (recording the order of sync ops) and the slave variant (replaying the order of sync ops). One consequence of this requirement is that our synchronization agents cannot dynamically allocate memory in the master variant, unless the slave variants perform the same memory allocations in the exact same order. This prevents us from using techniques like queue projection, as was done by Basile et al. [7].

4. Implementation

We implemented our design in two security-oriented MVEEs: GHUMVEE [43] and ReMon [45][4]. Our design is very general and can be ported to other MVEEs with minor effort. We will focus our discussion on ReMon, as it is an extended and more efficient version of GHUMVEE.

ReMon is a multithreaded monitor that targets x86 variants running on the GNU/Linux platform. Each of ReMon's threads monitors one set of equivalent variant threads. To initialize the MVEE, ReMon uses a bootstrap process to set up the variants, their respective monitors, and the shared ring buffers used by the monitors to communicate with each other. The bootstrap process hands over control to the monitors once the monitors and variants are fully initialized. After bootstrapping, the monitors operate exactly as described in Section 2. The monitors use two types of shared buffers. Specifically, they use ring buffers to (i) compare system call arguments, replicate system call results, and to coordinate the operation of the MVEE as a whole, and (ii) to capture and replay sync ops. To avoid confusion in the remainder of the paper, we will refer to the ring buffers used to monitor and replicate system calls as **syscall buffers**, and those used to capture and replay sync ops as **sync buffers**.

4.1 Ordering System Calls

ReMon enforces an equivalent system call ordering in all variants using Lamport's logical clocks [22]. Each monitor maintains a private copy of a logical clock, which we call the *syscall ordering clock*. This clock is used to assign timestamps to system calls that must be ordered. Whenever the master (leader) variant executes such a call, the monitor enters a critical section and records the current time on the syscall ordering clock into the syscall buffer. The critical section is not exited until the system call returns and the results and the timestamp have been written into the syscall buffer. Whenever a slave (follower) variant begins executing the same system call, its associated monitor will wait in a tight loop until the recorded timestamp matches the current time in the slave monitor's private copy of the syscall ordering clock. As soon as the timestamp matches the time of the clock, the monitor enters a critical section and resumes the variant to complete the system call. When the system call returns, the slave monitor increments the time of the syscall ordering clock and leaves the critical section.

Limitations. Our system call ordering mechanism wraps system calls in critical sections. As a result, we cannot order blocking system calls, because, on the one hand, the monitor does not exit the critical section until the blocking system

[4] The source code for ReMon, GHUMVEE, the synchronization agents, and the covert channels PoCs we present in this paper can be found online at `https://github.com/stijn-volckaert/ReMon`

call returns, and, on the other hand, it cannot guarantee that a blocking system call will return at all. Luckily, most blocking system calls that are subject to ordering in ReMon are those that perform I/O operations[5]. I/O operations are only executed by the master variant, and our monitor replicates the results of these operations to the slaves, thus guaranteeing that all variants receive consistent system call results. This eliminates the need to order these calls.

4.2 Ordering Sync Ops

Direct inter-thread communication is the main cause of nondeterminism in multithreaded variants. We eliminate this nondeterminism by enforcing an equivalent ordering of communicating instructions among all variants. As we limit our scope to data-race-free programs, it suffices to order synchronization operations, rather than each individual communicating instruction. To enable this ordering, we first identify and instrument the relevant sync ops at compile time. At run time, the instrumented program will call the synchronization agent before and after it executes each sync op to capture the sync op order in the master variant and to replay that same order in the slave variants.

4.3 Identifying Sync Ops

We developed a simple and efficient strategy for identifying sync ops in x86 programs. Our strategy is motivated by the fact that accesses to synchronization variables are, by definition, atomic. In x86 binary code, atomic accesses can only be expressed using one of the following types of instructions: (i) instructions with a `LOCK` prefix, (ii) `XCHG` instructions, (iii) aligned load/store instructions. Considering each instruction of any of these types to be sync ops, and instrumenting them as such, would be overly conservative. Typically, the vast majority of instructions of the latter type do not access synchronization variables at all, and imposing an order on such instructions would be a waste of CPU cycles.

Instead, we run a two-stage analysis on the program. In the first stage, we mark all instructions of types (i) and (ii) as sync ops. In the second stage, we run a points-to analysis and mark instructions of type (iii) as sync ops if and only if they may alias with variables pointed to by instructions of type (i) and (ii). We currently run the first stage of the analysis on the assembly code level using a Ruby script. The script marks all instructions of type (i) and (ii) and uses the debugging info in the program binary to map the instructions to their corresponding source lines. We performed the second stage of the analysis manually for the programs we evaluate in Section 5. However, we also built a tool that reduces the manual effort considerably as discussed in Section 4.3.1.

To see how this simple strategy works in practice, consider the code example in Listing 1. This listing shows a simplistic, yet valid implementation of a spinlock. The `compare_and_swap` function at line 4 is a compiler intrinsic

[5] The only exception is `sys_futex`, which we treat as an I/O operation, so the following reasoning still applies.

```
1  int spinlock;
2
3  void spinlock_lock(int* ptr) {
4    while(!compare_and_swap(ptr, 0, 1))
5      sched_yield();
6  }
7
8  void spinlock_unlock(int* ptr) {
9    *ptr = 0;
10 }
11
12 spinlock_lock(&spinlock);
13 // critical section
14 spinlock_unlock(&spinlock);
```

Listing 1. Ad-hoc implementation of a spinlock.

that emits a `LOCK CMPXCHG` instruction. This instruction is of type (i) and would therefore be considered a sync op in the first stage of the analysis. The subsequent points-to analysis would find that the store instruction at line 9 can point to a variable that is also pointed to by the sync op at line 4. It would therefore consider the instruction at line 9 to be a sync op as well.

Our strategy is sound, but not complete as there may be synchronization variables that are not pointed to by sync ops of type (i) or (ii). Yet, through extensive experimentation and evaluation, we found that the strategy suffices to support a large number of programs.

Limitations. Our approach to identifying sync ops has one major limitation: it does not work for synchronization primitives that rely solely on aligned load/store instructions. Consider for example the code in Listing 2. This example shows a naive implementation of a condition variable. Neither the load at line 9, nor the store at line 4 would have a `LOCK` prefix when compiled to assembly code. Our analysis would therefore fail to recognize either of these instructions as sync ops in the first analysis phase. Consequently, they would not be recognized by the second analysis phase either.

Fortunately, synchronization variables that are accessed only by aligned load/store instructions must be marked `volatile` for the code to be compiled correctly by an optimizing compiler. The `volatile` qualifier prevents the compiler from eliminating (e.g., with register allocation) or reordering accesses to the associated variable. An obvious extension of our analysis would therefore be to mark volatile variables as synchronization variables too, prior to running the points-to analysis. This extension would likely lead to an over-approximation of the number of synchronization variables. We expect this over-approximation to be minor, as volatile variables are rarely used for any purpose other than inter-thread synchronization in user-space programs. Note, however, that an analysis that identifies all uses of volatile variables cannot fully replace the analysis we described before, as non-volatile variables can still be used as synchronization variables, provided that they are only accessed from (inline) assembly code.

```
volatile int flag = 0;

void signal_thread() {
  flag = 1;
}

void wait_until_signaled() {
  while(!flag)
    sleep(1);
}
```

Listing 2. Unsupported synchronization primitive.

4.3.1 Automation Opportunities

Although we have manually performed the second phase of the analysis on the programs we tested, it is possible to automate the analysis to a large extent when the source code is available. We considered the following two ways to automate the identification process.

Whole-program alias analysis. Based on the feedback from our analysis' first stage, which is already fully automated, an inter-procedural alias analysis could, in theory, pinpoint the exact locations synchronization variables are accessed in the program. We have prototyped two tools in the LLVM compiler-framework to perform the identification of sync ops at the LLVM Intermediate Representation (IR) level. The first implementation leverages the DSA analysis framework in LLVM's `poolalloc` module [24]. The second implementation is based on the more recent `SVF` analysis framework [40]. Neither of these implementations currently yield satisfactory results when analyzing large code bases. In both cases, the majority of type (iii) instructions that target heap-allocated variables are classified as potential aliases of type (i) and (ii) instruction operands.

Our first implementation is based on data structure analysis which is a Steensgaard-style, unification-based points-to analysis [39]. Although DSA is field-sensitive, we found that the field sensitivity is often lost because heap objects of incompatible types get unified. The SVF analysis that we use in our second implementation is an Andersen-style, subset-based points-to analysis [2]. Although SVF does a better job at maintaining field sensitivity, we found no way to query its field sensitive results for heap objects. Furthermore, SVF is overly conservative when analyzing programs containing pointer arithmetic. We leave an in-depth exploration of these program analysis issues to future work as they are orthogonal to the challenges addressed herein.

Explicit type qualification. The second approach we have explored is based on the observation that clang, the front-end for C-related languages in the LLVM framework, translates all uses of variables that are explicitly marked with the `_Atomic` [17] type-qualifier into LLVM IR instructions that are explictly marked with an atomic flag. As the LLVM IR instruction set is agnostic to the atomicity guarantees of the target machine instruction set, this atomic flag is also applied to regular load and store instructions. Consequently, there

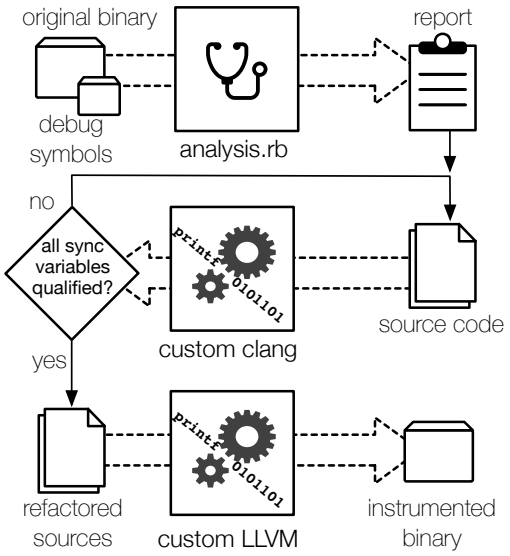


Figure 3. Explicit type qualification workflow.

is no need for a points-to analysis if the programmer expresses all synchronization operations using the atomic types and intrinsic functions defined by the C11 standard. The downside of this approach, compared to the aforementioned alias analysis-based approach, is that it requires source code refactoring. Furthermore, the C standard permits certain operations (such as casting to and from `void` pointers) that allow programmers to discard the `_Atomic` qualifier prior to accessing a variable.

We modified clang to guide refactoring of source code to only use explicitly qualified synchronization variables. Our modified version of clang imposes a stronger typing discipline onto the programmer by (i) displaying a warning whenever a pointer to a non-qualified variable is cast to a pointer to an `_Atomic`-qualified variable, (ii) displaying an error and terminating compilation whenever a pointer to an `_Atomic`-qualified variable is cast to a pointer to a non-qualified variable, and (iii) displaying an error and terminating compilation whenever an `_Atomic`-qualified variable is used in inline assembly code.

We use our tool as shown in Figure 3. First, we use a stock version of the LLVM compiler to compile the original, unmodified source code of the program into a binary with embedded debugging symbols. We analyze the resulting binary using the Ruby script we mentioned before. Based on the output of this script, we add type-qualifiers to variables used in sync ops. We then repeatedly compile the refactored code using our modified version of clang, and use clang's warnings and errors to propagate the `_Atomic` type-qualifier up and down the def-use chains of all pointers to sync variables until we reach a fixpoint where all sync variables, as well as pointers to sync variables are fully qualified. At this point, clang will stop displaying warnings, and it will translate the source code into LLVM IR code as intended. Finally, at the

```
1  void spinlock_lock(int* ptr) {
2    bool result = false;
3    while(!result) {
4      before_sync_op(ptr);
5      result = compare_and_swap(ptr, 0, 1);
6      after_sync_op(ptr);
7      if (result) break;
8      sched_yield();
9    }
10 }
11
12 void spinlock_unlock(int* ptr) {
13   before_sync_op(ptr);
14   *ptr = 0;
15   after_sync_op(ptr);
16 }
17
18 void __attribute__((weak)) before_sync_op(void* ptr) {}
19 void __attribute__((weak)) after_sync_op(void* ptr) {}
```

Listing 3. Instrumented spinlock.

IR level, we use a modified version of LLVM, described below, to add calls to our synchronization agent.

Although it has already proven to be useful in its current state, our tool can still be improved in several ways. First, we could extend the tool to assign the `_Atomic` qualifier automatically to volatile variables. As we argued before, the `volatile` qualifier can be used for synchronization variables that are only accessed using aligned load/store instructions, and are therefore not identified as sync ops by our script. Second, we could try to automate the refactoring. This might be difficult, however, since qualifying a pointer to a synchronization variable could affect multiple compilation units. Third, in certain cases, we could permit the use of `_Atomic` in easy-to-analyze inline assembly blocks.

4.4 Instrumenting Sync Ops

After identifying the relevant sync ops, the next step is to insert the calls to the synchronization agent. We wrap the sync ops at compile time, as shown in Listing 3. It shows the original program code in black and the instrumentation code in red. The `before_sync_op` and `after_sync_op` functions are implemented in the synchronization agent. In our current workflow, we dynamically link instrumented programs with the agent, which means that the agent must be loaded for the program to be able to run. This could be avoided by adding empty versions of these two functions as weak symbols to the instrumented program. This way, the program would call the agent, if it is running, or perform a no-op, if the agent is not running.

4.5 Synchronization Agents

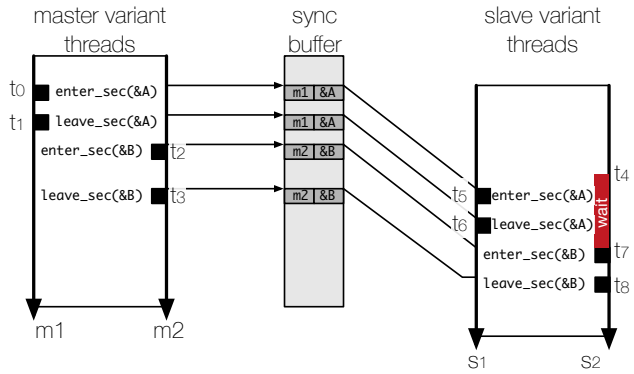
We developed three different agents that each implement one of the sync op replication strategies laid out below. All of them are shared libraries that implement the `before_sync_op` and `after_sync_op` functions. The MVEE forces the variants to load the agent by setting the `LD_PRELOAD` environment variable [26]. During its initialization, the agent attaches to the synchronization buffer using the System V IPC interface [27]. The agent records the order in which the master variant executes its sync ops in this synchronization buffer. The slave variants' agents can consult the data in this buffer to replay their sync ops in an equivalent order. To ensure that the master variant's agent knows that it must record sync ops and the slave variants' agents know that they must replay sync ops, we added a new system call that allows the variants to become self-aware. We did not have to patch the kernel to add this system call, as non-existing system calls are still reported to the MVEE's monitor.

Total-order replication agent. Our total-order (TO) replication agent replays all sync ops in the exact same order in which they happened in the master variant. Figure 4(a) shows two threads that execute under ReMon's control. In the master variant, thread m1 first enters and leaves a critical section protected by lock A at times t_0 and t_1 resp. At those times, the wrappers of the corresponding sync ops log the activities of thread m1 in the sync buffer. Next, thread m2 in the master variant enters and leaves a critical section protected by lock B at times t_2 and t_3 respectively. These events are also logged chronologically in the buffer. Right after t_3, the buffer holds the contents indicated in the figure. Time stamps to the left and right of the buffer mark the time the buffer elements are produced and consumed respectively. The arrows on the left (right) show the position of the producer (consumer) right after t_3.

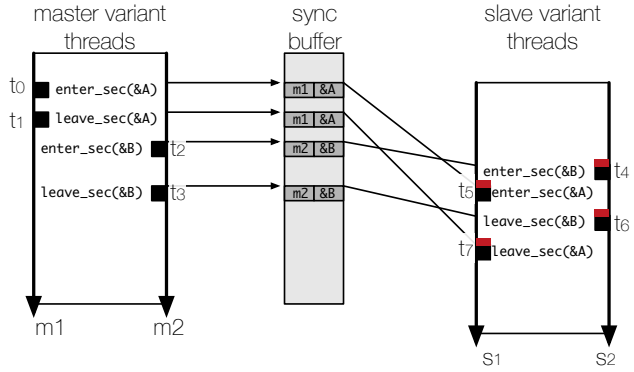
In the slave variant thread s2, corresponding to m2 in the master variant, reaches the critical section protected by lock B first, at time t_4. At that time, the first element in the buffer indicates that synchronization events in the master variant occurred in thread m1 first, so thread s2 is stalled in the wrapper of the sync op in enter_sec. Only after the first two elements in the buffer are consumed in thread s1 at times t_5 and t_6, can thread s2 continue executing. Thus, even though the two critical sections protected by locks A and B are unrelated, thread s2 is forced to stall until thread s1 has replayed the operations performed by thread m1.

This agent is trivial to implement, but not very efficient: The lack of lookahead by consumers introduces unnecessary stalls as indicated by the red bar in Figure 4(a).

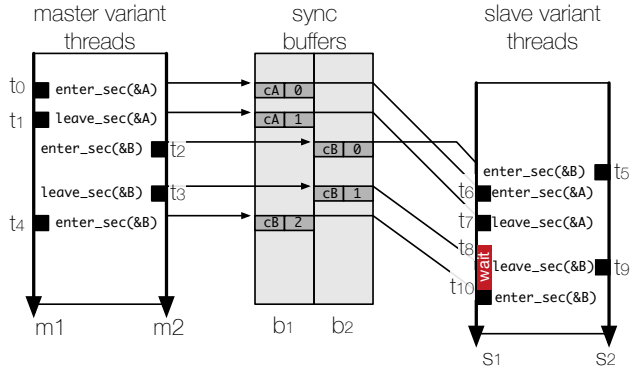
Partial-order replication agent. Our partial-order (PO) replication agent is more efficient with respect to stalling. It only enforces a total order on dependent sync ops. The sync ops are considered dependent if they operate on the same memory locations. This agent may replay independent sync ops in any order, as long as it preserves sequential consistency within the thread. The PO agent is more complex and introduces more memory pressure because the agents in the slave threads have to scan a window, containing information about the sync ops that have not been replayed yet, in the buffer to look ahead. However, it typically introduces much less stalling and generally outperforms the TO agent. In Figure 4(b), we see the exact same order of events as in Figure 4(a) until t_4. This time, however, thread s2 may enter



(a) Total-order replication



(b) Partial-order replication



(c) Wall-of-clocks replication

Figure 4. Replay sequences of three replication strategies.

the critical section without delay at t_4 because the enter_sec operation does not depend on either of the operations that preceded it in the recorded total order.

Although the PO agent eliminates unnecessary stalling, it still suffers from poor scalability. The master variant must safely coordinate access to the sync buffer by determining the next free position in which it can log an operation. If many threads simultaneously log operations, this inevitably leads to read-write sharing on the variable that stores the next free position. A similar problem exists on the slave variants' side because the threads within each variant must keep track of which data has been consumed by the variant. With multiple slave variants, this also leads to high sharing and, consequently, high cache pressure and coherency traffic.

Wall-of-clocks replication agent. The above observation led us to the design a third agent. This wall-of-clocks (WoC) agent assigns all synchronization variables (which are the subject of sync ops) to one of a fixed number of logical clocks. These clocks capture "happens-before" relationships between related sync ops [22].

In Figure 4(c), lock A stored at address &A is assigned to clock cA. Lock B is similarly assigned to clock cB.

On the master side, the agent logs the identifier of the logical clock associated with each sync op, as well as that clock's time. After logging each sync op, the agent increments the logical clock time of the associated clock.

In this agent, the logging is no longer done in a single sync buffer. Instead there is one sync buffer per master thread, such that each buffer has only one producer. In Figure 4(c), master thread m1 only communicates with slave thread s1 through buffer 1, whereas thread m2 only communicates with thread s2 through buffer 2. This design avoids cache contention when accessing shared buffers.

Neither the master nor the slave variants need to communicate their current buffer positions to other threads. Furthermore, the master's logical clocks do not need to be visible to the slaves. The information contained within the sync buffers is sufficient for the slave variants to replay the same clock increments on their own local copies of each clock.

In Figure 4(c), thread m1 first enters a critical section protected by lock A at time t_0. The agent observes that the current time on logical clock cA is 0. It records the clock and its time in buffer 1 and increments the clock's time to 1. At time t_1, the agent logs the exit from the critical section in buffer 1. This time around, the logical clock time is 1.

A similar situation then unfolds in thread m2 at time t_2. This time though, the critical section is protected by lock B, of which the associated memory location is assigned to clock cB, whose initial time also is 0. This information is logged in sync buffer 2, along with information regarding the exit of the critical section in thread m2 at time t_3. At that point, clock cB is incremented to 2.

In thread m1, a third critical section is entered at time t_4, which is again protected by lock B. This event involving logical clock cB is logged in buffer 1 with clock time 2.

On the slave variant's side, the threads are scheduled differently in our example. There, thread s2 reaches a sync op first, at time t_5. The agent observes in buffer 2 that it must wait until clock cB reaches time 0. Since this is the initial time on the slave's copy of that clock, the operation can be executed right away and thread s2 will increment the time on its copy of cB to 1. If we suppose that thread s2 is then pre-empted and thread s1 gets scheduled, s1 will enter and leave the critical section protected by lock A at times t_6 and t_7, consuming the first two entries in buffer 1, thereby incrementing the slave copy of clock cA to 2.

The third operation in thread s1 at time t_8 is the most interesting. In the first sync buffer, the slave agent observes that the sync op to enter a critical section has to wait until its associated logical clock cB has reached time 2. However, in the slave, that clock's time was last incremented at time t_5, i.e., to the value of 1. Thread s1 must therefore wait until some other slave thread has incremented the time on cB. This will happen at time t_9 in thread s2. Shortly thereafter, the agent code executing in thread s1 will observe that cB has reached the necessary value, and at t_{10} s1 will enter its second critical section.

With this WoC, the replication agent only inserts accesses to shared data, and hence coherence traffic, for two reasons. First, it introduces accesses to sync buffers shared between corresponding threads in the master and slave variants. This is a fundamentally unavoidable form of overhead required to replicate the synchronization behavior from the master to the slave variants.

Secondly, the agent inserts accesses to shared clocks whenever multiple threads in the original program were already contending for locks at shared memory locations. While these extra shared accesses in the replication agents still introduce some overhead, we do expect the overhead to scale with the pre-existing resource contention in the original application. In other words, if the original application uses contended, global locks that decrease the available parallelism, the replication agent will hurt it further. However, if the original application involves a lot of synchronization, but that synchronization is performed using uncontended, local locks, the WoC replication agent will not increase contention within the master or slave variants either.

As we will see in Section 5, the WoC agent consistently outperforms the other agents on almost every benchmark. Most importantly, as is the case with plausible clocks in general, the replication will always be correct [41].

One important remark remains to be made, however. While the WoC agent is certainly the more elegant and more efficient of the three proposed designs, it is not fully optimal. Our synchronization agents are prohibited from dynamically allocating memory, as we explained in Section 3.3. Thus, we cannot dynamically assign each memory location to its own private clock. Instead, we have to pre-allocate a fixed number of clocks statically and we have to assign lock memory locations to one of those clocks based on a hash of their memory address. Because we want to use a cheap hash function, hash collisions are quite likely. Any such collision results in an m-to-1 mapping between multiple locks and each clock. In other words, the WoC agent is bound to assign some non-conflicting memory locations to the same logical clock. When this happens, this introduces unnecessary serialization and hence potentially also unnecessary stalls in the slave variants. That said, our WoC agent purposely assigns unrelated sync variables to the same clock. This happens, for example, with adjacent 32-bit sync variables where the first one is aligned to a 64-bit boundary, because a single CMPXCHG8B x86 instruction could modify both variables at the same time.

4.5.1 Handling Diversity

Our synchronization agents tolerate any form of diversity that does not change the synchronization behavior of the program. If the variants do exhibit different synchronization behavior, which might happen, for example, if they use different memory allocators, then our synchronization agents will likely not function correctly. Our agents do, however, fully support address space layout diversity, without assigning logical identifiers to synchronization variables, and without maintaining an explicit mapping between the master and slave addresses for the same logical variables. If the n-th sync op executed by a master thread affects the variable at address &A, then the corresponding slave threads know based on the information recorded by the master, that their n-th sync op should affect the same logical variable, even if that variable is placed at a different addresses in the slave address spaces.

5. Evaluation

We evaluated three aspects of our proposed techniques. First, we evaluated the correctness and performance impact of our synchronization agents by running two synthetic benchmark suites under ReMon. Second, we evaluated the number of sync ops instrumented in the aforementioned benchmark suites. Third, we evaluated the security impact of the agents and the shared sync buffers. Finally, we tie everything together by modifying a realistic multithreaded server program to use our synchronization agents. We then run multiple diversified variants of this server program, evaluate the performance, and show that ReMon successfully detects attacks against the program.

5.1 Correctness and Performance Evaluation

We evaluated the correctness and performance impact of our techniques by running the PARSEC 2.1 and SPLASH-2x benchmark suites on top of ReMon. We analyzed and instrumented the programs in these suites and ran them with four worker threads. We excluded PARSEC's canneal and SPLASH-2x' cholesky benchmarks from our tests. The canneal benchmark is intentionally racy and therefore fundamentally incompatible with multi-variant execution environments such as ours and cholesky does not run correctly when compiled on our system (even outside the MVEE).

Several of the benchmark programs contained unintentional data races which were identified and patched in the literature [38]. We applied this set of patches, which were kindly shared by the author of the aforementioned work.

Run-time Overhead. We ran our benchmarks on a dual-socket server machine containing two Intel Xeon E5-2660 CPUs with 8 cores and 16 threads each. This machine has 64Gb of DDR3 RAM and 20MB of CPU cache per socket. To maximize the reproducibility of our results, we disabled hyper-threading, as well as all forms of power management

	2 variants	3 variants	4 variants
total-order agent	2.76x	2.83x	2.87x
partial-order agent	2.83x	2.83x	3.00x
wall-of-clocks agent	1.14x	1.27x	1.38x

Table 1. Aggregated average slowdowns for each of the synchronization agents

and clock frequency scaling. To isolate the performance impact of our synchronization and replication mechanisms, we disabled Address Space Layout Randomization (ASLR) and did not apply any diversity techniques to our variants[6]. Our server ran Ubuntu 14.04.04 LTS with version 3.13.11 of the Linux kernel. We used `glibc` 2.19 and compiled all software with `gcc` 4.8.

Figure 5 shows the performance results for PARSEC and SPLASH respectively. We also show aggregated averages for each synchronization agent in Table 1. We measured the native run time by running the non-instrumented binaries outside our MVEE. We performed two sets of measurements for single-variant executions[7] of the benchmarks. For one set, we enabled both physical CPUs. For the other set, we enabled just one of the physical CPUs. We included only the best results for each benchmark in Figure 5. Benchmark programs that execute few system calls but many sync ops (e.g., PARSEC's streamcluster benchmark) ran significantly faster with one CPU disabled. This is likely due to the operating system's scheduler's tendency to balance the system load by spreading the benchmarks' threads across both CPUs, which significantly increases the cost of cache misses on a NUMA machine. A higher cost of cache misses, in turn, increases the overhead incurred by our synchronization agents.

We compare the performance of the single-variant results with the performance of two up to four variants of the instrumented binaries running inside the MVEE and tested with all three of our synchronization agents. For all of our tests, we used the largest available input set. We kept both of the physical CPUs enabled for all of the MVEE benchmarks since this was the most favorable configuration.

We observe several trends in these results. First, our partial-order agent outperforms the total-order agent in most benchmarks, because the former introduces fewer unnecessary stalls. The total-order agent still performs better on average, however, because the cache contention problems we described in Section 4.5 clearly manifest in `radiosity`, `fluidanimate`, and `swaptions` (with two variants), and in `dedup`, and `ocean_ncp` (with more than two variants). The wall-of-clocks agent generally outperforms both agents.

A second observation is that, in some benchmarks, we see superlinear performance degradation when moving from three variants to four variants. This is because the total number of simultaneously executing threads exceeds the number

[6] We did run separate tests with ASLR and diversity techniques enabled, as discussed later on.

[7] Native executions of the benchmarks without using an MVEE.

	Benchmark	run time	syscall rate	sync rate
PARSEC 2.1	blackscholes	80.83	2.55	0.00
	bodytrack	60.06	8.59	202.36
	dedup	18.29	134.27	1052.45
	facesim	142.52	4.14	288.75
	ferret	103.79	2.29	225.10
	fluidanimate	93.19	0.45	12746.59
	freqmine	168.66	0.35	0.24
	raytrace	147.54	0.78	88.33
	streamcluster	136.05	5.63	18.78
	swaptions	86.68	0.01	4585.65
	vips	37.09	15.76	428.69
	x264	34.73	0.50	15.98
SPLASH-2x	barnes	61.15	19.61	5115.99
	fft	40.26	0.01	1.64
	fmm	42.68	0.91	5215.01
	lu_cb	51.16	0.08	0.23
	lu_ncb	73.55	0.05	0.16
	ocean_cp	39.39	1.21	5.05
	ocean_ncp	41.68	1.08	4.55
	radiosity	45.56	33.42	18252.68
	radix	18.22	0.02	0.04
	raytrace	52.52	6.63	536.79
	volrend	52.02	15.86	1071.25
	water_nsquared	182.80	0.88	8.61
	water_spatial	59.84	148.27	9.63

Table 2. Native run times (in sec), and system call (in 1000 system calls/sec) and synchronization operation (in 1000 sync ops/sec) rates for PARSEC and SPLASH runs using four worker threads.

of physical CPU cores our system has available. `dedup` and `ferret`, for example, are pipelined benchmarks that run $3*n$ and $2+4*n$ threads respectively in parallel[8] Similarly, `vips` runs $2 + n$ threads in parallel.

Finally, we observe that several benchmarks perform badly, even when sufficient system resources are available. `dedup`, for example, is $1.78\times$ slower relative to its native speed when running two variants with the wall-of-clocks agent, while `barnes` and `radiosity` are $1.61\times$ and $1.47\times$ slower under the same conditions. These slowdowns can, in part, be attributed to the high number of system calls and/or sync ops these benchmarks execute. As Table 2 shows, `dedup` executes over 134K system calls 1.02M sync ops per second, whereas `barnes` and `radiosity` execute more than 19K and 33K system calls per second and 5.12M and 18.25M sync ops per second resp. Each of the system calls invokes the MVEE monitor, which constitutes a performance bottleneck even in the most efficient security-oriented MVEEs [21, 45].

Correctness We verified the correctness of our proposed techniques by repeating the above tests with ASLR enabled

[8] n is the number of worker threads, which was 4 in our tests.

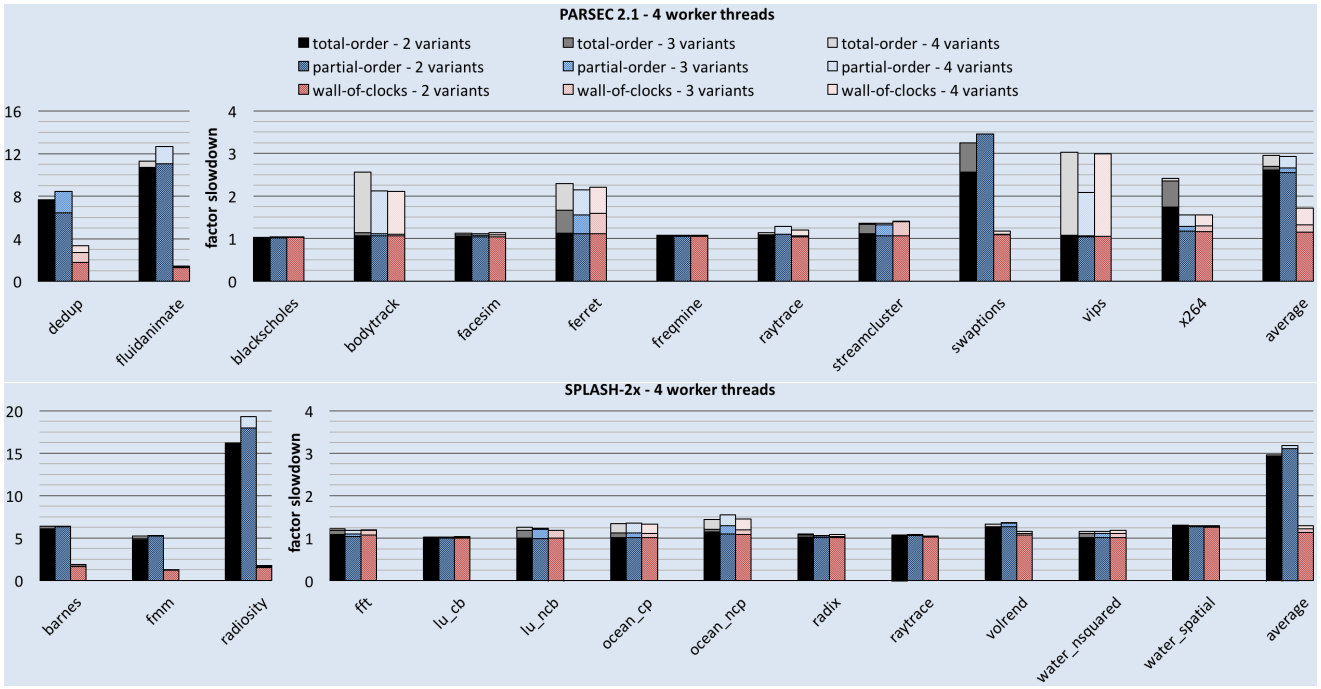


Figure 5. Run-time performance overhead in PARSEC and SPLASH, relative to native execution. The three stacks per benchmark correspond to the three synchronization agents.

and with the non-overlapping code technique proposed in related work applied to our variants [44]. We tested a variety of monitoring policies ranging from strict lockstepping on all system calls to lockstepping only on security-sensitive system calls. Our monitor is configured to detect divergence under each of these configurations. No divergence was detected in any of the benchmarks indicating that our proposed techniques function correctly. Although the performance impact of the diversity techniques we have tested is generally small ($<$1%), there are some benchmarks in which these techniques introduce a non-negligible overhead. Since the focus of this paper is replication of sync ops, we disabled diversity in our performance measurements.

5.2 Sync Ops Analysis

We ran the two-stage analysis described in Section 4.3 on all of the benchmark programs. Table 3 shows an overview of the sync ops we identified in each of the benchmarks and in the shared libraries against which they are linked. We omitted all programs and libraries in which we did not find any sync ops. Type (i) and (ii) instructions were automatically identified using a disassembler in the first stage of the analysis. Type (iii) instructions were identified based on the output of the first stage, and a manual points-to analysis.

It is worth noting that not all of the shared libraries are used by all of the benchmarks. `libgomp`, the runtime library for `OpenMP` programs, is used only in PARSEC's `freqmine` benchmark. Similarly, `libstdc++`, the runtime library for `C++` programs, is only used in PARSEC's `streamcluster`, `bodytrack`, `fluidanimate`, `raytrace`, `facesim`, `swaptions`, and `freqmine`.

	(i)	(ii)	(iii)
Base Libraries			
`libc-2.19.so`	319	409	94
`libpthreads-2.19.so`	163	81	160
`libgomp.so`	68	38	13
`libstdc++.so`	162	3	25
PARSEC 2.1 binaries			
`bodytrack`	201	0	8
`facesim`	385	0	8
`raytrace`	170	0	8
`vips`	4	0	6

Table 3. Sync ops identified in the PARSEC and SPLASH benchmark suites. Type (i)/(ii) sync ops are instructions with explicit/implicit LOCK prefixes. Type (iii) sync ops are aligned load/store instructions to variables that are accessed by type (i) or (ii) instructions elsewhere in the program.

The effective number of type (iii) instructions is likely higher than the numbers we report in Table 3. Since our analysis operates at the source-code level, we do not account for compile-time inlining. The instrumentation we describe next is still correct, however, because we perform the instrumentation at the source level too.

5.3 Instrumenting Synchronization Operations

Although automation is possible, we instrumented all of the sync ops we identified in the aforementioned programs and libraries manually. The instrumentation was straightforward for `libgomp` and `libstdc++`. These libraries can be built for a generic (i.e., non-x86) target. The source code for the generic target does not contain any inline assembly

code, and does not make any assumptions about the atomicity of aligned memory operations. As such, all sync ops in the generic source tree use the easily identifiable `gcc`'s atomic intrinsics [42], and regular memory accesses to synchronization variables (i.e., the type (iii) instructions in Section 5.2) are wrapped in `atomic_load` and `atomic_store` instrinsics. We therefore chose to instrument these libraries by overriding the atomic intrinsics at preprocessing time.

Instrumenting `libc` and `libpthreads` required more effort. Much of the code in these two libraries predates the introduction of atomic intrinsics in `gcc`. All of the sync ops in `libc` and `libpthreads` are therefore expressed in target-specific assembly code. Moreover, type (iii) instructions are (at least in code that targets the x86 platform) not explicitly marked in the code as the developers implicitly assumed that aligned loads and stores are atomic.

The sync ops we identified in PARSEC's `vips` benchmark originated from GNOME's base library `libglib`, which is linked into the `vips` binary. Rather than to instrument the sync ops directly, we configured `libglib` to use pthread sync primitives rather than its own sync ops.

The sync ops we identified in PARSEC's `bodytrack`, `facesim`, and `raytrace` binaries were in code inlined from the Standard Template Library (STL). The inlined code performs thread-safe reference counting on STL containers. Although accesses to the reference counter variables should be instrumented in the general case, we did not have time to instrument them in these benchmarks. Despite not instrumenting these sync ops, we did not observe any divergent behavior in these benchmarks. Furthermore, we believe that the performance gain from not instrumenting the sync ops is negligible. Specifically, the non-instrumented sync ops account for only $6.3\text{e}{-6}\%$, $3.9\text{e}{-7}\%$, and 0.0016% of the total number of instructions executed by `bodytrack`, `facesim`, and `raytrace` respectively.

5.4 Security Analysis

To understand the implications of supporting multi-threaded applications on the MVEE's security guarantees, we take a closer look at the components we added to ReMon: the synchronization agent and the synchronization buffer. The sync agent runs at the same privilege level as the rest of the variant's code. Thus, it cannot interfere with the monitor's core tasks. We can therefore limit our analysis to the implications of having a sync buffer mapped into the variants' address spaces.

This sync buffer is mapped onto writable memory pages that are shared among all variants. These pages are not hidden, nor isolated from the variants in any way. However, our monitor does ensure that each buffer is mapped at different, non-overlapping addresses in all variants. The buffer could therefore, in theory, be used to provide input that is not regulated by the monitor to all variants. However, our system is designed such that only the synchronization agents ever access the sync buffer, and such that no data in the buffer is ever copied to any other memory location. Thus, as long as the variants have not been compromised, the buffer does not pose any additional channel to deliver exploit payloads.

Now, while our extensions do not alter the security guarantees of the underlying MVEE, it *is* possible for malicious programs to communicate pointer values across different variants by abusing the replication mechanism. This breaks the underlying security assumption for MVEEs in general that the variants cannot communicate private data such as pointer values with one another and with the outside world.

The main observation is that replicating the results of a system call across variants can result in a covert channel. For example, the result of `sys_gettimeofday` and the `rdtsc` instruction are replicated from the master variant to the slave variants. If a variant has a data-dependent delay between two system calls, the time delta between the execution of the second and first call is data-dependent. This data-dependent delta is then replicated to the other variant, which can recover the data from the delta. To abuse this successfully, both variants need to be 'self-aware': one variant needs to receive data while the other variant sends, and vice versa. One can probabilistic decide whether a variant is the master or slave by having each variant hash a pointer value, which will differ across the variants. The variants then decide whether or not to send based on this hash. At the end of the exchange, both variants have the randomized pointer values of both themselves and the other variant. Both variants can send these values to an outside attacker without any divergence being detected.

A similar covert channel can be made by abusing the replication of synchronization primitives to send data from the master variant to the slave variant. In an application with two threads, the first thread can lock and unlock a mutex, where the unlocking happens after a data-dependent loop. This data-dependent delay allows us to force whether or not a `pthread_mutex_trylock` call on that mutex will succeed in the second thread. Whether or not the `trylock` succeeds will be replicated in the slave variant, which can again reconstruct the original data by observing the pattern of succeeded and failed lock attempts.

These proof of concept covert channels show that variants can communicate with each other and then leak their randomized data to the outside world, despite the common argument that monitors will detect any memory-dependent values being sent to the outside world by different variants. However, we stress that this is *not* an issue with our extensions; rather, it is an issue with MVEEs in general. Furthermore, it is unclear how such covert communication and the ability to send/receive variant-specific pointer values can be used to mount a useful, concrete attack on a realistic program rather a malicious proof of concept program. This seems to require either very specific code flaws and code patterns, or a degree of control over the program's execution, such as a JIT-based attack, that would make this covert channel super-

fluous. Mounting such an attack is outside the scope of this security analysis.

5.5 Use Case: nginx

Now that we have evaluated all the performance, security, and correctness aspects of our MVEE and its synchronization agents in isolation, we want to show that our findings are still valid when we run a realistic setup. We chose to run version 1.8 of the nginx web server, which recently introduced a thread pooling feature [5]. Part of the inter-thread synchronization in nginx is based on `pthread` synchronization primitives, which we had already covered as they are widely used in the synthetic benchmarks we tested too. On top of these primitives, the nginx developers have also built some synchronization primitives of their own, using inline assembly code and compiler intrinsics.

As we expected, if we do not instrument these custom synchronization primitives, nginx does not function correctly when running multiple variants on top of our MVEE. The server does start up normally, but quickly triggers a divergence when network traffic starts flowing in. We then analyzed, refactored, and instrumented the variants using the tools described in Section 4.3.1. This whole process took less than fifteen minutes. We identified 51 sync ops in total in the nginx configuration we tested.

We verified that instrumented variants run correctly on top of our MVEE, even with Address Space Layout Randomization (ASLR), Disjoint Code Layouts (DCL) [44], and Position Independent Executables (PIE) enabled, by generating web requests using the wrk benchmark tool. To evaluate the server performance, we ran the wrk tool on a client machine that communicated with our server via a local gigabit network. We used the same dual-socket machine we mentioned before to run two instrumented variants of nginx with ASLR and DCL enabled on top of ReMon. nginx was set up to spawn thread pools with 32 threads. We set wrk up to generate requests for a static 4KiB web page using 10 simultaneous connections over a period of 10 seconds. The average throughput of the variants running in the MVEE was 3% lower than the native throughput of the non-instrumented program. We then repeated this test with the network traffic sent on a loopback interface (and the benchmark tool running on the server itself). In this case, the average throughput inside the MVEE was 48% lower than the native throughput.

Finally, we verified that our MVEE still detects divergence when the server is under attack. To do so, we constructed a code-reuse attack that exploits the CVE-2013-2028 vulnerability (which we re-introduced in this version of nginx), and used a script to dynamically tailor the attack to a specific running victim variant of nginx. As expected, we found that our attack could successfully compromise nginx running natively, or running as a single variant inside our MVEE. When running two or more variants, however, our MVEE detected divergence and shut down all variants before the system could be compromised.

6. Related Work

Multithreading in MVEEs. GHUMVEE and ReMon are not the only MVEE that have facilities specifically meant to support multithreaded programs. With Orchestra, Salamat et al. were the first to claim support for multithreaded programs [37]. Orchestra uses one monitoring thread, each responsible for monitoring one set of variant threads, to compare the variants' system call sequences per-thread rather than globally. This design allows Orchestra to tolerate the differences in the global system call sequences that naturally occur in all types of multithreaded programs due to differences in how each variant's threads are scheduled.

GHUMVEE, our own MVEE that formed the basis for ReMon, was the first to extend support to multithreaded programs in which the threads interact with each other [43]. GHUMVEE forces all of the variants' system calls to execute in lockstep, and prevents system calls operating on shared resources to execute simultaneously. The first version of GHUMVEE also supported inter-thread synchronization, but only came with the total-order synchronization agent. GHUMVEE originally also required fully manual identification and instrumentation of synchronization operations.

With VARAN, Hosek and Cadar presented the first MVEE that eschews lockstepping in favor of loose variant synchronization [19]. They proposed to use logical clocks to capture the order in which the leader variant executes its system calls, and to replay that same order in the follower variants. This design suffices to support loosely-coupled multithreaded variants, but fails when the variants use explicit inter-thread synchronization through shared memory.

Deterministic Multithreading (DMT). DMT systems impose a deterministic schedule on the execution order of instructions that participate in inter-thread communication, or a deterministic schedule on the order in which the effects of those instructions become visible to other threads. Some DMT systems guarantee determinism only in the absence of data races (*weak determinism*), while others work even for programs with data races (*strong determinism*).

Some DMT implementations, especially the older ones, rely on custom hardware [8, 15, 16, 20] or a custom operating system [4, 10]. Of interest to us, however, are the user-space software-based approaches [6, 9, 11, 14, 15, 28, 29, 31, 32, 34, 46]. Software-based DMT systems come in many flavors but essentially, they all establish a deterministic schedule by passing a token. We refer to the literature for an excellent overview of the possible ways to implement the deterministic schedule, as well as their implications [38]. In the remainder of this discussion, we focus on the fundamental reason why DMT systems are incompatible with MVEEs that run diversified variants: the timing of and prerequisites for the deterministic token passing.

Most DMT systems require that all threads synchronize at a global barrier before they can pass their token. Some of the systems that employ such a global barrier, insert calls to

the barrier function only when a thread executes a synchronization operation [6, 11, 28, 34]. This approach is incompatible with parallel programs in which threads deliberately wait in an infinite loop for an asynchronous event such as the delivery of a signal to trigger. Such threads never reach the global barrier. Other DMT systems tackle this issue by inserting barriers at deterministic points in the thread's execution. These deterministic points are based on the number of executed store instructions [32], the number of issued instructions [46] or the number of executed instructions [9, 15]. All of these numbers are extremely sensitive to small program variations, which makes such systems an ill fit for use in diversified variants.

Conversion [31] does not use a global barrier but, like other DMT systems, it relies on a deterministic token that can only be passed when threads invoke synchronization operations, which again is incompatible with parallel programs in which some threads never invoke synchronization operations. RFDet [29] uses an optimized version of the Kendo algorithm [32] to establish a deterministic synchronization order. Like Kendo however, the order is still based on the amount of executed instructions in each thread, which makes RFDet equally sensitive to program variations.

Record/Replay (R+R). R+R systems capture the order of synchronization operations in one execution of a program and then enforce the same order in a different execution. This can happen offline, by capturing the order in a file to be replayed during a later execution of the same program, or online, by broadcasting the order directly to another running instance of the program. In the absence of data races, R+R systems show many similarities with DMT techniques that impose weak determinism.

RecPlay is a prime example of an offline R+R system [35]. During recording, RecPlay logs Lamport timestamps for all pthread-based synchronization operations [22]. During subsequent replay sessions, synchronization operations are forced to wait until all operations with a earlier timestamp have completed. Because it only enforces the order of synchronization operations, RecPlay's replication mechanism incurs less overhead than preexisting techniques that replicate the thread scheduling order or the order in which interrupts are processed [3]. Moreover, RecPlay assigns the same timestamp to non-conflicting synchronization operations, such that they can also be replayed in parallel.

Loose Synchronization Algorithm (LSA) was one of the first techniques that adopted R+R for use in fault-tolerant systems [7]. LSA designates one of the nodes as the master node. The master node records the order of all pthread-based mutex acquisitions and periodically replicates this order to the slave nodes. These slave nodes then enforce the same acquisition order on a per-mutex basis.

More recently, Lee et al. proposed Respec online replay on multi-processor systems [25]. Oriented towards fault-tolerant execution of identical variants, Respec purposely records an unprecise order of synchronization operations in the master process and speculatively replays that order in the slave processes. At the end of a replay interval, Respec checks whether the slaves are still synchronized with the master process by comparing their state, incl. their register contents. If not, it rolls them back. While recording, Respec maps synchronization variables onto a statically allocated clock, similarly to our wall-of-clocks agent. It is doubtful, however, whether Respec's approach could work in a security-oriented MVEE like ours, in which diversity in the variants makes it hard (if not impossible) to detect whether the variants have diverged at the end of a replay interval.

Other online R+R techniques rely on custom hardware support [8], and hence are not useful for a secure MVEE for off-the-shelf systems.

7. Conclusion

Multi-variant execution environments increase the resilience of systems software by forcing adversaries to simultaneously compromise multiple, diversified program variants without causing divergence. This makes MVEEs a great fit for legacy applications where security concerns justify the additional resource consumption. Unfortunately, lack of support for multi-threaded applications severely limits the use cases for MVEEs at a time where even low-end, mobile devices contain multi-core processors.

Our paper evaluates three synchronization strategies. One of these, our novel wall-of-clocks agent demonstrates that partially ordering synchronization operations among threads while avoiding cache contention minimizes the overhead of protection (1.14x and 1.38x for 2 and 4 variants respectively). Additionally, we proposed a new strategy to embed a replication agent into parallel programs, including programs that use ad hoc synchronization primitives, and we described the effort to do so. We believe that, with additional engineering effort, this strategy can be automated to a large degree.

Acknowledgments

The authors thank Jean-Pierre Lozi, our reviewers, the Agency for Innovation by Science and Technology in Flanders (IWT), and the Fund for Scientific Research - Flanders.

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124 and FA8750-15-C-0085, by the National Science Foundation under award numbers CNS-1513837 and CNS-1619211, as well as gifts from Oracle and Qualcomm. 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 (DARPA), its Contracting Agents, or any other agency of the U.S. Government.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proc. of the 12th ACM conference on Computer and communications security*, pages 340–353, 2005.

[2] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[3] K. M. Audenaert and L. J. Levrouw. Interrupt replay: a debugging method for parallel programs with interrupts. *Microprocessors and Microsystems*, 18(10):601–612, 1994.

[4] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. *Communications of the ACM*, 55(5):111–119, 2012.

[5] V. Bartenev. Thread pools in nginx boost performance 9x! `https://www.nginx.com/blog/thread-pools-boost-performance-9x/`, 2015.

[6] C. Basile, Z. Kalbarczyk, and R. Iyer. A preemptive deterministic scheduling algorithm for multithreaded replicas. In *Proc. IEEE Int'l Conf. Dependable Systems and Networks*, pages 149–158, 2002.

[7] C. Basile, Z. Kalbarczyk, and R. Iyer. Active replication of multithreaded applications. *IEEE Trans. on Parallel and Distributed Systems*, 17(5):448–465, 2006. ISSN 1045-9219.

[8] A. Basu, J. Bobba, and M. D. Hill. Karma: scalable deterministic record-replay. In *Proc. Int'l Conf. on Supercomputing*, pages 359–368, 2011.

[9] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. *ACM SIGARCH Computer Architecture News*, 38(1):53–64, 2010.

[10] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *Proc. OSDI*, pages 177–192, 2010.

[11] E. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. *ACM Sigplan Notices*, 44 (10):81–96, 2009.

[12] L. Cavallaro. *Comprehensive Memory Error Protection via Diversity and Taint-Tracking*. PhD thesis, Univ. Degli Studi Di Milano, 2007.

[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. 15th USENIX Security Symp.*, pages 105–120, 2006.

[14] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: a practical runtime for deterministic, stable, and reliable threads. In *Proc. ACM Symp. Operating Systems Principles*, pages 388–405, 2013.

[15] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. *ACM SIGARCH Computer Architecture News*, 37(1):85–96, 2009.

[16] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: a relaxed consistency deterministic computer. *ACM SIGPLAN Notices*, 46(3):67–78, 2011.

[17] I. O. for Standardization. Iso/iec 9899:2011: C11 standard, 2011.

[18] P. Hosek and C. Cadar. Safe software updates via multi-version execution. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 612–621, 2013.

[19] P. Hosek and C. Cadar. VARAN the unbelievable: An efficient n-version execution framework. In *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 339–353, 2015.

[20] D. R. Hower, P. Dudnik, M. D. Hill, and D. A. Wood. Calvin: Deterministic or not? free will to choose. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 333–334, 2011.

[21] K. Koning, H. Bos, and C. Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2016.

[22] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7):558–565, 1978.

[23] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy (SP)*, pages 276–291, 2014.

[24] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *ACM Sigplan Notices*, volume 40, pages 129–142, 2005.

[25] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. *ACM SIGARCH Computer Architecture News*, 38(1):77–90, 2010.

[26] Linux Programmer's Manual. ld.so(8)-Linux Manual Page, .

[27] Linux Programmer's Manual. shmat(2)-Linux Manual Page, .

[28] T. Liu, C. Curtsinger, and E. Berger. DTHREADS: efficient deterministic multithreading. In *Proc. ACM Symp. on Operating System Principles*, pages 327–336, 2011.

[29] K. Lu, X. Zhou, T. Bergan, and X. Wang. Efficient deterministic multithreading without global barriers. In *Proc. 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 287–300, 2014.

[30] M. Maurer and D. Brumley. Tachyon: Tandem execution for efficient live patch testing. In *USENIX Security Symposium*, pages 617–630, 2012.

[31] T. Merrifield and J. Eriksson. Conversion: Multi-version concurrency control for main memory segments. In *Proc. ACM European Conf. on Computer Systems*, pages 127–139, 2013.

[32] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. *ACM Sigplan Notices*, 44(3):97–108, 2009.

[33] PaX Team. PaX non-executable pages design & implementation. `http://pax.grsecurity.net/docs/noexec.txt`, 2004.

[34] H. Reiser, J. Domaschka, F. J. Hauck, R. Kapitza, and W. Schröder-Preikschat. Consistent replication of multithreaded distributed objects. In *Proc. IEEE Symp. Reliable Distributed Systems*, pages 257–266, 2006.

[35] M. Ronsse and K. De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Trans. on Computer Systems*, 17(2):133–152, 1999.

[36] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proc. ACM Conf. on Programming language design and implementation*. ACM, 1996.

[37] 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. EuroSys Conf.*, pages 33–46, 2009.

[38] C. Segulja and T. S. Abdelrahman. What is the cost of weak determinism? In *Proc. Int'l Conf. Parallel architectures and compilation*, pages 99–112, 2014.

[39] B. Steensgaard. Points-to analysis in almost linear time. In *Proc. 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 32–41. ACM, 1996.

[40] Y. Sui and J. Xue. SVF: interprocedural static value-flow analysis in llvm. In *Proc. 25th International Conference on Compiler Construction*, pages 265–266. ACM, 2016.

[41] F. J. Torres-Rojas and M. Ahamad. Plausible clocks: constant size logical clocks for distributed systems. *Distributed Computing*, 12(4):179–195, 1999. ISSN 0178-2770.

[42] Using the GNU Compiler Collection (GCC). atomic builtins. `https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html`, 2016.

[43] S. Volckaert, B. De Sutter, T. De Baets, and K. De Bosschere. GHUMVEE: efficient, effective, and flexible replication. In *Proc. Int'l Symp. on Foundations and practice of security*, pages 261–277, 2013.

[44] S. Volckaert, B. Coppens, and B. De Sutter. Cloning your gadgets: Complete ROP attack immunity with multi-variant execution. *IEEE Trans. on Dependable and Secure Computing*, 13(4):437–450, 2015.

[45] S. Volckaert, B. Coppens, A. Voulimeneas, A. Homescu, P. Larsen, B. D. Sutter, and M. Franz. Secure and efficient application monitoring and replication. In *USENIX Technical Conference*, pages 167–179. USENIX, 2016.

[46] X. Zhou, K. Lu, X. Wang, and X. Li. Exploiting parallelism in deterministic shared memory multiprocessing. *Journal of Parallel and Distributed Computing*, 72(5):716–727, 2012.