A DSL Approach to Reconcile Equivalent Divergent Program Executions

Luís Pina Daniel Grumberg Anastasios Andronidis Cristian Cadar
Department of Computing
Imperial College London, UK
{*l.pina, daniel.grumberg14, a.andronidis15, c.cadar*}*@imperial.ac.uk*

Abstract

Multi-Version Execution (MVE) deploys multiple versions of the same program, typically synchronizing their execution at the level of system calls. By default, MVE requires all deployed versions to issue the same sequence of system calls, which limits the types of versions which can be deployed.

In this paper, we propose a Domain-Specific Language (DSL) to reconcile expected divergences between different program versions deployed through MVE. We evaluate the DSL by adding it to an existing MVE system (Varan) and testing it via three scenarios: (1) deploying the same program under different configurations, (2) deploying different releases of the same program, and (3) deploying dynamic analyses in parallel with the native execution. We also present an algorithm to automatically extract DSL rules from pairs of system call traces. Our results show that each scenario requires a small number of simple rules (at most 14 rules in each case) and that writing DSL rules can be partially automated.

1 Introduction

Multi-version execution (MVE) has seen a revival in recent years as a mechanism to increase software security and reliability [13, 18, 20, 22, 29, 34, 35]. At a high-level, MVE works by running multiple versions of a program in parallel, synchronizing their execution typically at the level of system calls. In a security context, one can run diversified program variants (e.g., where each variant has a different memory layout) in such a way that divergences across variants signal a security attack [29, 34]. In a reliability context, one can run diversified variants or multiple software revisions and allow the overall application to continue execution when versions crash [18, 19].

In its initial instantiation, MVE employs a monitor process that intercepts all the system calls issued by the underlying versions. When all versions issue the same system call, the monitor executes the system call once on behalf of all versions, and copies the results to each version. If any version diverges, i.e. issues a different system call, the monitor raises a warning and stops executing (in a security context) or terminates the divergent versions and MVE continues with fewer versions (in a reliability context).

There are two main issues with this simple form of MVE. First, executing system calls from all versions in lock-step imposes a large performance penalty. Second, this form of MVE relies on all versions issuing the *same* sequence of system calls. The latter issue is particularly problematic because it limits the types of versions that can be run with MVE. For instance, the diversified variants cannot issue different but equivalent sequences of system calls (e.g., those arising due to refactoring), and the MVE system cannot ignore additional system calls (e.g., that one version may use for extra logging).

A new architecture, recently introduced by Varan [19], tackles both issues. In the proposed scheme, which resembles an in-memory record-replay framework, there is no central monitor. Instead, one of the versions acts as the *leader* and executes system calls directly, writing their results into a shared ring buffer. The other versions, *followers*, simply read back the results from the ring buffer (faster followers always wait for the leader). In terms of performance, Varan allows the leader to run at almost native speed, as it does not require the leader to synchronize with the followers. While Varan provides flexibility in terms of matching the sequences of system calls issued by different versions, it does not provide an easy expressive way to encode the differences in system call sequences that should be tolerated across versions.

In this paper, we propose a simple, elegant, and expressive domain-specific language (DSL) specifically designed for writing system call matching rules that allows a follower to reconcile its sequence of system calls with that of the leader (§3). We show that this DSL allows the use of MVE in a wider range of scenarios with minimal effort, requiring only a small number of rules in each case. In particular, we show the applicability of our approach with three different MVE scenarios: (1) running versions of the same application with different configurations (§2.1), where we needed only 7 rules to execute Redis under multiple configurations (§5.2), e.g., with and without a persistent store; (2) running different software revisions (§2.2), where we required only 7 rules to run versions of Redis which are up to 730 commits apart (§5.3); and (3) running native versions of an application in parallel with versions instrumented for dynamic analysis (§2.3), where we needed only 14 rules to support the Valgrind tool [24], 3 rules to support Asan [30], 1 rule to support Msan [33], and 4 rules to support Tsan [31] (§5.4).

We also provide an empirical evaluation that shows that simply comparing pairs of strace logs, which list the sequence of system calls that each version issues when run in isolation, is enough to write all the DSL rules (§5). No knowledge about the particular MVE system or the versions being used is needed to write the DSL rules. Inspired by how we manually found the rules, we provide an algorithm to synthesize some of the rules based on such pairs of strace logs (§4).

In summary, we make the following contributions:

1. The first paper to present a simple solution to the problem of handling divergent executions in MVE, which allows MVE to be easily applicable to many more scenarios, such as running an application concurrently under different configurations; running different releases of the same program; and running native versions in parallel with versions instrumented for dynamic analysis.
2. The design and implementation of a small and expressive DSL that encodes rules to handle divergences, and our experience using it in the three scenarios described above.
3. The design and implementation of an algorithm that synthesizes part of the DSL rules using pairs of strace logs, which can be obtained by running each version in isolation over the same inputs.
4. An empirical evaluation of our prototypes for the Varan MVE, that shows the applicability of each scenario and provides evidence about the little effort required to write the rules, and how much this task can be further automated by the DSL synthesis algorithm.

2 Applicable Scenarios

At a high-level, some program executions can be considered equivalent even if they do not execute the same code. As a trivial example, two executions of the same correct deterministic C program under different memory allocators can be considered equivalent because their *observable behavior*—the sequence of system calls they issue—is the same. However, there are scenarios in which it is beneficial to deploy programs with MVE that issue different sequences of system calls. For instance, one may increase reliability by deploying two releases of the same program [18] in which the order of some system calls are changed, but without affecting the overall behavior of the program—e.g., one release may simply change the order in which two files are opened.

In this paper, we describe a domain specific language (DSL) designed to easily encode and tolerate such divergences, and thus enable many useful MVE scenarios. In the rest of this section, we present and motivate three scenarios that can take full advantage of our DSL.

2.1 Different Configurations

Depending on its configuration parameters, software can behave differently by enabling or disabling features such as logging. For instance, Redis[1] is an in-memory key-value store that can optionally dump the store to persistent storage periodically or after every request.

There are three scenarios in which running different program configurations under MVE can be useful. First, for increased reliability: Different configurations may trigger different bugs so running several configurations in parallel increases the chance of at least one configuration staying alive and providing service. Second, for increased security: If security is critical, one may choose to stop as soon as any configuration diverges in its core execution from the others. The rationale here is that an attack may succeed in one configuration, but not all, as different configurations have slightly different memory layouts, issue different sequences of system calls, etc. Third, for inexpensive logging and error diagnosis: A fast configuration (no logging, no debugging info, full compiler optimizations, etc.) can be deployed at full speed, as the leader, while slower configurations (with logging, debugging, etc.) can be deployed in the background as followers.

Different configurations share the core functionality of the program, but each implements additional features such as logging and persistent storage. From the perspective of their external behavior, the sequence of system calls issued by an expensive configuration is typically a superset of the base configuration. For instance, the Redis configuration that adds persistence issues extra system calls to open the persistent file on disk and write data into it. In particular, one will see additional calls as below, interleaved with the core functionality of the program:

[1]`https://redis.io/`

```
...
open("persistentStore", ...) = 7
...
write(7, ...) = 10
..
write(7, ...) = 45
...
```

As we show in §3, our DSL makes it easy to encode such divergences, allowing MVE systems to run multiple configurations of the same program concurrently.

2.2 Different Software Releases

MVE is an effective technique to increase the reliability of software updates [11,18]. Instead of updating the software to a new version that becomes available, the idea is to run both the new and the old version in parallel. If one version fails, the system can revert to the other version. This technique mitigates the problem of unreliable software updates [14,28,36], as the old version is still available in the background in case the new version crashes.

Mx [18] applied this approach successfully, but it could deploy only versions that issue the same sequence of system calls. However, as we show in this paper, tolerating certain classes of system call divergences allows one to handle a much wider range of software updates.

In general, the external behavior of the software is stable, especially in mature applications. However, small changes in the sequence of system calls occur even for mature applications. Examples include: (1) slightly changing the API used, and (2) changing the order in which some system calls are performed. As an example in the first category, Lighttpd revision 2436 changes its sequence call sequence from `geteuid`, `geteguid` to `geteuid`, `getuid`, `getegid`, `getgid` [19]. As an example in the second category, Redis version 2.0.1 reorders the sequence `setsockopt`, `time`, `epoll_ctl` into `setsockopt`, `epoll_ctl`, `time`.

Our DSL makes it easy to express such differences. As we show in §5.3, we were able to run together Redis versions up to 730 commits apart while only using a small number of simple DSL rules.

2.3 Native and Sanitized Versions

Dynamic analysis techniques instrument or interpret the program under analysis to detect common programming errors. For instance, Asan [30], the address sanitizer that ships with modern C compilers, instruments memory buffers in the program with red zones to detect buffer overflow errors. Valgrind [25], a dynamic analysis tool that takes program binaries as input, interprets the program and shadows all the memory that the program uses to detect a large category of bugs, such as buffer overflows and invalid uses of uninitialized memory.

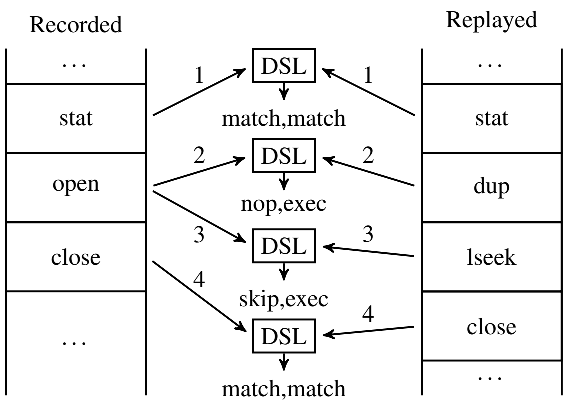

Figure 1: Example of reconciling two divergent sequences of system calls: `open` on the recorded side matches `dup` and `lseek` on the replayed side.

MVE can be used to deploy sanitized versions in the background [19]. The key idea is to run the native version of the program as a leader—this is the version that interacts with users and runs at full speed. Sanitized versions are then run as followers, checking the execution for errors in the background.

One of the main challenges involved is that the sanitized versions change the sequence of system calls that the program under analysis issues. For instance, analyses may use signal `SIGSEGV` internally (e.g., to allocate more shadow memory). This signal may also be used by the program under analysis. In this case, the analysis technique registers its own signal handler and intercepts attempts from the program under analysis to register another signal handler, through system call `rt_sigaction` in 64-bit GNU/Linux, by forwarding signals generated outside the analysis to the program's handler. We describe some of the changes in further detail in §3.1.

As we show in §5.4, our DSL is able to encode the divergences introduced by real dynamic analysis techniques, such as Asan [30], Msan [33], Tsan [31] and Valgrind [25], using only a small number of easy-to-write rules.

3 DSL for Reconciling MVE Divergences

We now propose a simple and expressive domain specific language (DSL) for describing system call divergences between two executions. Our design is driven by real-world examples illustrating the scenarios described in §2.

Figure 1 shows the high-level architecture of the DSL we propose. The DSL operates between two sequences of system calls: the *recorded* and the *replayed*. At each step, and for each sequence, the DSL takes as input the next system call and generates as output the ac-

```
dsl  ::= [#include "file"] [rule]
rule ::= [lhs-syscall] => [rhs-syscall]
       | begin => [rhs-syscall]
       | group g = { [ name ( [arg] ) ] }
lhs-syscall ::= scall pred lbl | nothing
rhs-syscall ::= scall pred callback
              | scall * | label | nothing

scall ::= name ( [arg] ) | g
pred  ::= { C-code } | ε
lbl   ::= as label | ε
callback ::= -> ( ret ) { C-code }
           | -> { C-code } | ε
name  ::= read | write | ...
arg   ::= _ | var
```

Figure 2: Syntax of the DSL. All words in bold and symbols besides `|`, `[`, and `]` are terminals. Square brackets denote possible empty comma-separated lists. `var`, `ret`, `label`, `file` and g are identifiers.

```
open(_,_,_) => dup(_), lseek(_,_,_)
```

(a)

```
rt_sigaction(_,_) as segv,
rt_sigaction(_,_) as ill,
rt_sigaction(_,_) as bus,
rt_sigaction(_,_) as fpe =>
 ill, rt_sigaction(_,_), bus, fpe, segv
```

(b)

```
group calls = { read(_,_,_), write(_,_,_) }

calls as self =>
 gettid(), write(_,_,_), rt_sigprocmask(_),
 self,
 rt_sigprocmask(_), gettid(), read(_,_,_)
```

(c)

```
open(p,_,_) {
  return !strcmp($(p), "overcommit");
} , read(_,_,_), close(_) => nothing
```

(d)

```
#include "globals.h" // declares ign

nothing => open("log.txt",_,_)
    -> (ret) { ign = $(ret); }

nothing => write(fd,_,_)
    { return $(fd) == ign; }
```

(e)

```
write(_,_,_) => write(_,_,count)
    -> (ret) {$(ret) = $(count)}
```

(f)

```
begin =>
 mprotect(_), mprotect(_) * , munmap(_)
```

(g)

Figure 3: Examples of DSL rules.

tion to take. For each matching system call between the recorded and replayed sequences (steps 1 and 4 in Figure 1), the DSL simply matches both sides through action MATCH, thus advancing both sequences by one position. In the example shown, taken from Valgrind, system call `open` is rewritten as a sequence of `dup` and `lseek`. The DSL reconciles the divergence in steps 2 and 3 through actions EXEC on the replayed sequence, to execute those system calls. In step 2, the recorded sequence is left unchanged through action NOP, while in step 3 it is advanced without matching anything on the replayed side through action SKIP.

3.1 Syntax

The syntax of the DSL is given by the grammar shown in Figure 2. A DSL input file is a collection of *rules*. Each rule defines how a sequence of recorded system calls, on the *left-hand side (LHS)* of the rule, matches a different sequence of replayed system calls, on the *right-hand side (RHS)* of that rule.

For instance, Figure 3a shows a rule that tolerates the divergence presented in Figure 1. The underscore characters allow any values for the respective arguments, so the rule matches a recorded `open` with a replayed sequence of `dup` and `lseek`, regardless of any arguments. For system calls where all arguments are unconstrained, we sometimes use a single underscore for brevity.

The RHS of each rule can refer to system calls on the LHS through *labels*. For instance, different releases of Redis register a different number of signal handlers in different order. The rule in Figure 3b shows how to use labels to reconcile such divergences.

Valgrind wraps 19 different system calls with the same three system calls before and three after. The user thus needs to repeat the same rule for each wrapped system call. Instead, the DSL supports *groups*: syntactic sugar to repeat the same rule for different system calls. Figure 3c shows an example, adapted from Valgrind, that groups all the system calls that use the same rule.

Some rules apply only when particular values are passed to those system calls at runtime. For instance, when the memory allocator *malloc* reaches a certain percentage of the available memory, it tunes its behavior based on the kernel overcommit settings by reading file `/proc/sys/vm/overcommit_memory`. The analyses Asan and Tsan increase the virtual memory to a large percentage of the whole available addressing space, which prevents the allocator from ever tuning its behavior. The rule in Figure 3d reconciles such executions between native and sanitized versions, through a *predicate written as C code*.

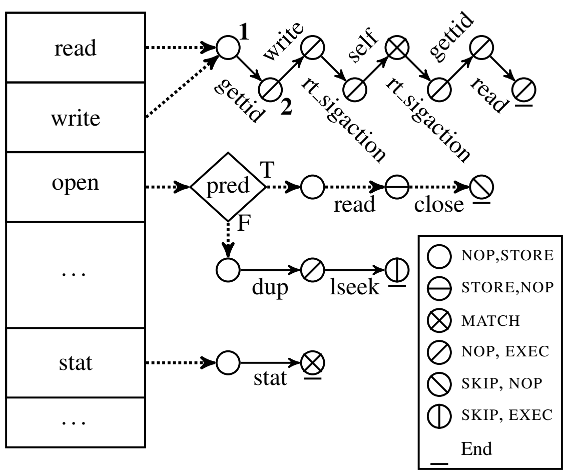

Figure 4: Visual representation of the DFAs generated for the rules in Figures 3c, 3d, and 3a, in that order. System call `stat` shows the default rule. Dotted and solid arrows transition on the LHS and RHS system calls, respectively. Node shapes denote actions performed on the recorded and replayed sequences on each transition.

The rule in Figure 3d also shows how to handle extra system calls that the replayed side issues without any correspondent on the recorded side through keyword `nothing` on the RHS. Of course, keyword `nothing` can also be used on the LHS. For instance, when deploying two versions of Redis in which the replayed side performs logging that the recorded side does not, we need a rule to ignore opening and writing to the log file. Figure 3e shows two such rules for `open` and `write`.

The rule in Figure 3e also shows an example of syntactic sugar for the common case of argument variables being literal strings or numbers. In this case, the DSL simply expands such a rule to an equivalent rule that uses predicates, similar to the rule shown in Figure 3d.

Predicates execute before a system call is matched, and therefore cannot access the results of that system call. Rules can also have a *callback* written in C, that executes after matching the system call and can access the (potentially modified) arguments and the return value. For instance, the rules shown in Figure 3e save the file descriptor of the matched `open` system call in global variable `ign`, which the rule for `write` then uses to discard the appropriate system calls. Note that the DSL allows the inclusion of C header files, which enables the rules to access libraries.

Predicates and callbacks can modify the arguments passed to the system call, and callbacks can also modify the return value. For instance, a version of Redis writes more bytes due to a protocol change. The rule shown in Figure 3f tolerates such a divergence by changing the return of the offending `write` on the replayed side.

The last part of the DSL, the `begin` rule, expresses a pattern that denotes the end of a large divergent prefix on the replayed side. System call matching using the other rules between both sides only starts after the pattern in the `begin` rule has been matched (which is empty by default). For instance, Valgrind sets up its internal state before starting to execute the program under analysis. This is when Valgrind sets handlers for interesting signals such as `SIGSEGV`, as discussed earlier. Valgrind finishes its set-up with one or more `mprotect` calls and a single `munmap` call. The rule in Figure 3g tolerates this large divergence in a compact way. This last example also shows the usage of the *star modifier (*)*, applied to the second `mprotect`, which matches the preceding system call zero or more times.

3.2 Semantics

The rules are implemented by a collection of *Deterministic Finite Automata (DFAs)*. Figure 4 shows the DFAs generated for some of the examples discussed in §3.1.

The algorithm starts by matching the current recorded system call with the first LHS on a rule to select a DFA. System calls that do not appear on the LHS of any rule have a default rule to themselves (e.g., rule `stat(_, _) as self => self` shown in Figure 4).

Rules are chosen in the same order in which they are defined. For instance, Figure 4 defines two rules that apply to system call `open`: the rules in Figures 3d and 3a, in that order. If the predicate is true, the algorithm chooses the first rule, otherwise it chooses the second one.

With a DFA selected, the algorithm uses it to match the rest of the sequence of system calls on the LHS of the rule, if any, with the sequence on the RHS. The DFA takes each system call to be reconciled, and either accepts it by moving to the next state, or rejects it. When the DFA rejects a system call, we say that an *irreconcilable divergence* has occurred because the two executions have diverged in a way that the DSL cannot reconcile. The DFA finishes once it accepts the final system call of a rule. The algorithm then discards the current DFA and uses the next recorded system call to select the next rule.

Rules that have `nothing` as the LHS are implemented as an exception to sequences rejected by the DFA. When the DFA rejects the first RHS system call of a rule with a single LHS, the algorithm then looks for a rule with `nothing` as the LHS that start with the offending RHS system call. If found, the algorithm follows that DFA instead of diverging. Otherwise, it diverges as described above. Rules with `nothing` on the LHS thus have lower precedence than all the other rules. Rules that have `nothing` as the RHS generate a DFA that only takes recorded system calls, as shown in Figure 4 for system call `open` when the predicate is true.

3.3 Interface with the MVE System

We now describe how an MVE system interacts with the DSL, using the example in Figure 1. Initially, the MVE system uses function `init` with the next recorded system call (e.g., `stat`) to choose a DFA. As suggested by Figure 4, a lookup table maps the first recorded system call to the DFA that implements the corresponding rule.

The MVE system then uses function `reconcile` to pass each recorded and replayed system call to the DSL. This function takes the current DFA, validates the next transition, and returns: the next DFA state, the actions to perform on the recorded and replayed sides, and some flags. On our running example, calling `reconcile(stat, stat)` yields actions MATCH on both sides. The MVE thus matches the two system calls and their arguments, copying the results from the recorded to the replayed side, and advances both sequences by one. Note that this behavior is what the MVE system does during regular operation without our DSL. This call also returns a flag that signals the end of this rule, so that the MVE uses function `teardown` to clean the resources of the finished DFA.

Following Figure 1, the next recorded call is `open`. Again, the MVE system uses function `init` to select the next rule. In this case, there is a choice between the rules in Figures 3d and 3a (defined in this order), depending on the truth value of the predicate in 3d. Let us assume that the predicate for 3d returns false, thus selecting the rule in 3a. At this point, calling `reconcile(open, dup)` yields actions NOP and EXEC on the recorded and replayed sides, respectively. Action NOP does nothing on the recorded side, while action EXEC executes the replayed system call without matching it with the recorded side. The MVE thus allows the replayed side to execute system call `dup` directly and calls `reconcile` with the same recorded call, `open`, and the next replayed call, `lseek`. Calling `reconcile(open, lseek)` returns actions SKIP and EXEC, for the recorded and replayed side, respectively. Action SKIP advances the recorded side one position, effectively ignoring the system call.

Let us now consider that the predicate for `open` returns true, selecting the rule in Figure 3d. In this case, calling `reconcile(open, read)` returns actions STORE and NOP for the recorded and replayed side, respectively. Action NOP on the replayed side means that the MVE system calls `reconcile` with the same replayed call, just as it does for NOP on the recorded side. Action STORE on the recorded side is useful for rules with multiple LHS calls, and prompts the MVE to advance the recorded side and call `reconcile` with the next recorded call. Later MATCH actions may refer to previous calls on the recorded side on which action STORE was taken, which means that the MVE needs to save all such recorded calls. For instance, the rule in Figure 3b returns the following sequence of actions for the recorded side: STORE, STORE, STORE, MATCH *2*, NOP, MATCH *3*, MATCH *4*, and MATCH *1*. Note that function `init` implicitly performs action STORE on the recorded side.

Function `reconcile` returns a special flag when an irreconcilable divergence occurs. The MVE must handle such a divergence, by reconciling it in some other way, stopping execution, or terminating that replayed version.

4 Automatic Synthesis of DSL Rules

When designing the DSL and writing rules for the different scenarios, we used *strace*,[2] an utility that logs all the system calls that a process issues, to generate system call traces for different program versions. For instance, we used this approach to compare the sequences of system calls issued by native and Valgrind versions of the same application when run on the same inputs.

We then noticed that a simple visual diff tool (*vimdiff*, part of VIM[3]) was able to display the two files side-by-side with most of the matching system calls aligned. Figures 5a and 5b show an example of such an aligned diff result. This provides empirical evidence that the rules are often easy to identify.

Based on our experience of manually writing the rules, we decided to create an automatic synthesis algorithm which targets the most common set of rules that we encountered, those of the form shown in Figure 3c, which wrap a system call with zero or more system calls before and zero or more system calls after. Ignoring grouping, 61% of the rules needed for Valgrind had this form.

Figure 5c shows the pseudo-code of the rule synthesis algorithm.[4] Function `synthesize` takes as input two system call traces, `lhs` and `rhs`, and returns a set of candidate rules. A candidate rule `Cand` is a triple `(before, s, after)` which defines the rule `s => before s after`, where `s` is a system call, and `before` and `after` are (possibly empty) sequences of system calls.

The algorithm iterates over each unique system call `s` in `lhs` (line 10). It aligns both sequences on the next instance of `s` on line 11, and creates an *initial candidate* `cand` by taking the `n` system calls around the aligned `s` on `rhs`. For instance, for `s=open` and `n=6`, the algorithm aligns both logs on position 56 and proposes rule: `open as s => access, getpid, getpid, gettid, write, rt_sigprocmask, s, rt_sigprocmask, gettid, read, fstat, mmap, fstat`.

[2]`https://strace.io/`

[3]`http://www.vim.org`

[4]For space reasons, the pseudo-code ignores error handling, particularly when function `align` fails.

```
50 access("ld.so.preload")
51
52
53
54
55
56 open("/etc/ld.so.cache")
57
58
59
60 fstat(3, ...)
61 mmap
62
63
64
65
66 close(3)
67
68
69
70
71
72 open("/lib/libm.so.6")
73
74
75
76
```

(a)

```
50 access("ld.so.preload")
51 getpid
52 getpid
53 gettid
54 write
55 rt_sigprocmask
56 open("/etc/ld.so.cache")
57 rt_sigprocmask
58 gettid
59 read
60 fstat(3, ...)
61 mmap
62 fstat
63 readlink
64 stat
65 mmap
66 close(3)
67 getpid
68 getpid
69 gettid
70 write
71 rt_sigprocmask
72 open("/lib/libm.so.6")
73 rt_sigprocmask
74 gettid()
75 read
76
```

(b)

```
1  // Helper functions:
2  // uniqueSCalls([[b,b,s,s,a,a,a] = [a,b,s]
3  // split(s, [b,b,s,s,a,a] = ([b,b], s, [s,a,a])
4  // takeHead([a,b,c,d],3) = [a,b,c]
5  // takeTail([a,b,c,d],3) = [b,c,d]
6  // intersectHead([a,b,c], [a,b,z]) = [a,b]
7  // intersectTail([b,c,d], [z,c,d]) = [c,d]
8
9  synthesize(Trace lhs, Trace rhs, int n) -> [Cand]
10  for s in uniqueSCalls(lhs) :
11   (lhs‘, rhsBefore,  rhs‘) = align(s, lhs, rhs)
12   before = takeTail(rhsBefore, n)
13   after = takeHead(rhs‘, n)
14   cand = (before, s, after)
15   cands[s] = refineCand(cand, lhs‘, rhs‘)
16  return cands
17
18 align(SCall s, Trace lhs, Trace rhs)
19       -> (Trace, Trace, Trace)
20  (_, l, lhsAfter) = split(s, lhs)
21  while (rhs != []):
22   (rhsBefore, r, rhsAfter) = split(s, rhs)
23   if (l.args == r.args) :
24     return (lhsAfter, rhsBefore, rhsAfter)
25
26 refineCand(Cand c, Trace lhs, Trace rhs) -> Cand
27  while (lhs != [] and rhs != []):
28   (lhs, rhsBefore, rhs) = align(c.s, lhs, rhs)
29   before = intersectTail(c.before, rhsBefore)
30   after = intesectHead(c.after, rhs)
31  return c
```

(c)

Figure 5: Example sequences of system calls issued by a native execution (5a) and the same execution under Valgrind (5b), and pseudo-code for the synthesis algorithm (5c). Matching system calls in 5a and 5b are aligned and highlighted.

Of course, the initial candidate rule is unlikely to be correct. The algorithm then refines that candidate using the rest of the logs on line 15 as follows. First, it finds the next aligned pair of the same system call on line 28. In our example, this yields position 72 in the traces. The algorithm then computes the intersection of the current rule with the system calls that surround the new matching on lines 29–30. The algorithm repeats this refinement step for each aligned pair of the same system call, iteratively discarding system calls that were captured by accident by the initial candidate. Back to our example, the algorithm discards positions 50 and 60–62, thus finding the correct rule: `open as s => getpid, getpid, gettid, write, rt_sigprocmask, s, rt_sigprocmask, gettid, read`.

In this case, the algorithm found the correct rule in a single refinement step, but this may not be always the case. For instance, if position 76 contained system call `fstat`, as position 60, then the algorithm would keep system call `fstat` as the end of the refined rule, resulting in an incorrect rule due to *over-capture*. The algorithm is prone to over-capture for under-represented system calls (e.g., those that only appear once or twice in the whole log) because the algorithm cannot refine them past the initial candidate(s). Sorting system calls by their frequency in function `uniqueSCalls` improves the quality of the results by leaving under-represented system calls to the end.

The algorithm is also prone to *misalignment*, when it aligns two system calls incorrectly and then generates the trivial rule `syscall` **`as`** `s => s`. In our experience, misalignment happens only due to *non-determinism* (e.g., user input timing). Note that the system call comparison, in line 23, already handles some non-determinism. For instance, two `write` system calls on the same file descriptor and with the same size are considered equal, regardless of the contents. Similarly, two `open` system calls in directory `/tmp` are considered equal, even if the files have different names. This allows to align executions that print the current time or the process ID, and executions that create temporary files with different names. Note also that a correct MVE system handles these and other sources of non-determinism that happen during runtime.

The algorithm also fails when the pattern for reconciling a given system call changes. This may happen based on the arguments passed to the system call (e.g., opening a special file uses a different rule). The DSL provides C predicates to handle such cases, but the synthesis algorithm cannot generate them.

5 Evaluation

This section describes the empirical evaluation of the DSL we propose for reconciling system call divergences across program executions. In particular, we evaluate the DSL for each of the application scenarios that we describe in §2, using Varan as the underlying MVE system [19]. We also describe the empirical evaluation of the DSL synthesis algorithm in generating rules for the sanitized versions scenario.

5.1 Implementation

We implemented the DSL parser and the synthesis algorithm in Haskell, with 1388 and 422 LOC, respectively.

We evaluated the DSL with Varan as the MVE system, whose architecture we presented in the introduction. We modified Varan to work with the DSL in several ways. First, Varan builds C files from DSL input files and includes the generated files during compilation. At runtime, we added a flag for Varan to load a particular DSL file for the execution. The DSL-based matching runs on the followers. In our experiments, we use a single follower, but in principle we could run multiple followers, each with its own DSL rules.

We include the generated C files with Varan at build time for ease of implementation. There is no fundamental reason to prevent each DSL file to be compiled separately (e.g., as a shared object) and loaded dynamically. This would make Varan easier to extend to other scenarios and we plan to implement this feature in the future.

Currently, we implemented rules with multiple LHS system calls through several rules with a single LHS, bound together through predicates that keep track of the matching sequence. In the future, we plan to implement this directly in the DSL.

5.2 Different Configurations

We explored the scenario in §2.1 by deploying Redis 3.2.6 with Varan using different configurations. The leader was configured to keep an in-memory store and write minimal logs. The follower used one of the following configurations: (1) persistent store,[5] (2) verbose (debug) logs, and (3) both 1 and 2. We required only 7 rules to handle all divergences.

As expected, the sequences of system calls issued in these three configurations were a superset of those issued by the leader. Most of the DSL rules simply ignored extra operations performed over file paths. For instance, Configuration 1 required a variation of the rule shown in Figure 3e to ignore manipulating the persistence file.

[5]`appendonly yes` and `appendfsync always`.

Table 1: Redis versions tested, number of commits between versions, and number of rules needed.

ID	Versions	Commits	Rules
1	1.3.8* – 1.3.10	40	0
2	1.3.10 – 1.3.12	105	0
3	1.3.12 – 2.0.0	92	1
4	2.0.0 – 2.0.5	34	1
5	2.0.5 – 2.2.0	730	3
6	2.2.0 – 2.2.15	110	2

* Revision `a71f072`

We wrote a C library to simplify managing ignored file paths and associated descriptors to minimize the C code needed for each rule.

Configuration 1 issues one less `gettimeofday` call early in the execution. To reconcile this divergence, we had to use a long rule that captures context from the previous 8 system calls. Configuration 2 issues strictly more `gettimeofday` to write timestamps on log entries. A simple rule `nothing => gettimeofday(_,_)` sufficed to tolerate such divergences. Configuration 3 simply required a trivial merge of the DSL files for Configurations 1 and 2. However, we note that in general, merging DSL files is not guaranteed to tolerate the combination of behaviors that each file tolerates in isolation.

5.3 Software Releases

As discussed in §2.2, the DSL can be used to deploy different program releases. We deployed the pairs of Redis versions listed in Table 1, by running the old version as the leader and the new version as the follower. We configured leader and follower to use separate log files, with a verbose logging level. We then added rules for ignoring log files, with 6 rules totaling 15 lines. These rules are common to all experiments and are not included in Table 1. We used the *redis-benchmark* included in Redis 1.3.8 as our workload, configured with a single client and performing one request for each operation (we are interested in functionality rather than performance).

We start with Redis 1.3.8 revision `a71f072` so that our results can be compared with Mx, which could not deploy different versions that change the sequence of system calls [18]. Pairs 1 and 2 required no additional rules.

In Pair 3, version `2.0.0` registers one more signal handler than previous versions (for `SIGTERM`), which can be expressed with the simple rule `nothing => rt_sigaction(15,_,_)`. In Pair 4, version `2.0.5` changes the order of a `time` system call, from before to after an `epoll_ctl`. We used a rule similar to the rule shown in Figure 3b to tolerate this divergence.

Pair 5 had most changes, with 730 commits. However, all divergences introduced in these commits required only three rules, similar to those presented before: ignoring file paths and `stat` calls on those paths (one rule); issuing extra `time` system calls on one side (one rule); and a `write` system call that writes more bytes than previous versions, due to protocol differences, as we described in detail in §3.1 and Figure 3f (one rule).

Finally, pair 6 required the rule `nothing => gettimeofday(_,_)` to handle extra system calls, and a rule to tolerate a change in the order of multiple `rt_sigaction` system calls, as shown in Figure 3b.

We were able to deploy six pairs with releases up to 730 commits apart with minimal effort (7 rules in total). Our approach works especially well for applications that keep backwards-compatibility, such as Redis, which tend to retain external behavior between releases and newer versions still support older data formats and protocols.

5.4 Dynamic Analyses

We used Varan to deploy the following existing dynamic analyses as followers of a native leader: Asan, Msan, Tsan, and Valgrind (with the memcheck tool). Asan [30], Msan [33], and Tsan [31] are the address, memory, and thread sanitizer, respectively, which ship with modern releases of popular C/C++ compilers Clang and GCC. We used the ones that ship with Clang version 3.8. Valgrind [32] checks for uses of invalid memory (i.e. uninitialized, unallocated, or freed memory) in C/C++ programs through heavyweight dynamic instrumentation. We used Valgrind version 3.11 built from revision 15920 (VEX revision 3233).

We executed Git[6] version 2.9.2, a widely-used version control system, with all the analyses described above (commands `log`, `blame`, `diff`, and `tag`). We also executed the following applications with Asan and Valgrind: `ssh` and `ssh-keygen` from OpenSSH[7] version 7.1, a suite of utilities used to secure communications by encrypting network traffic; HTop[8] version 2.0.1, an interactive system monitor and process viewer, and VIM version 7.4, a screen-oriented text editor.

We manually wrote a DSL file for each analysis, making it possible to run all the programs we mentioned with all the configurations we listed. Msan required the smallest DSL file, with 1 rule totaling 7 lines; Asan required 3 rules totaling 10 lines, and Tsan required 4 rules totaling 13 lines. The most interesting rule, shared by all these three analyses, is shown in Figure 3d and described in §3.1. Other rules ignore the system calls in which the leader sets up signal handlers for signals that the analyses already handle, as described in §2.3; and ignore extra system calls that the analyses issue (`nanosleep` and `sched_getaffinity`).

As expected, Valgrind required more effort with 14 rules totaling 104 lines. Valgrind required a `begin` rule with 3 lines to ignore its initialization (rule in Figure 3g). We also grouped 19 system calls that use one of two rules (one shown in Figure 3c), thus saving implementation effort and improving the readability of the DSL file.

Handling system call `open` under Valgrind required two rules: a general rule, illustrated by positions 51–59 in Figure 5; and a specialized rule for file `/proc/self/cmdline`, the one in Figure 3a but with an appropriate predicate. When a process attempts to read the command line that launched it, Valgrind hides the fact that the process is being run under analysis by treating that `open` system call in a particular way instead of interpreting it directly. The specialized rule appears before the general rule in the file and, as explained in §3.3, has higher precedence.

We also needed some rules with `nothing` as the LHS. For instance, when the application under analysis attempts to `mmap` a file into memory, Valgrind issues more system calls to allocate adequate shadow memory for that file. Given that Varan does not copy the `mmap` system call to the ring buffer, we tolerated the divergence between positions 61–65 in Figures 5a and 5b with the following rule:

```
nothing => mmap(_), fstat(_,_,_),
  readlink(_,_), stat(_,_), mmap(_)
```

Other rules with `nothing` as the LHS skip extra work that Valgrind performs to schedule threads, and when the program under analysis loads a dynamic library.

5.5 Synthesis Algorithm

All the rules described so far in the experimental evaluation were manually written by comparing sequences of strace logs side-by-side, as shown in Figure 5. We evaluated the DSL synthesis algorithm by using similar strace logs as input, to infer the rules needed to tolerate divergences between native and Valgrind executions, and comparing them with the ones that we wrote manually. We used the workloads for Git, OpenSSH, and VIM described in §5.4. We also used four GNU/Linux command-line utilities: `ls` and `du` from `CoreUtils`[9] version 8.25, `grep`[10] version 3.0, `cal` from `util-linux`[11] version 2.29.2, and the DSL synthesis algorithm itself. The algorithm took under 22 seconds on a modern laptop to run on each pair of traces.

[6]https://git-scm.com/
[7]http://www.openssh.com/
[8]http://hisham.hm/htop/

[9]https://www.gnu.org/software/coreutils
[10]https://www.gnu.org/software/grep/
[11]https://www.kernel.org/pub/linux/utils/util-linux/

Table 2: Rules synthesized from pairs of native and Valgrind strace logs, including partial and incorrect rules due to under-represented system calls (^{a}), over-capture (^{b}), or misalignment (^{c}).

Program	Rules	Correct	Partial	Incorrect
git tag	4	4	0	0
git diff	5	4	1^a	0
git log	5	5	0	0
ls	6	5	1^b	0
grep	5	5	0	0
cal	4	4	0	0
du	5	4	1^a	0
keygen	5	4	1^b	0
ssh	9	6	1^a	2^c
synth	6	6	0	0
vim	11	6	2^b	3^c

Table 2 shows the results for the 19 different rules that can be synthesized from these traces (this is the number of rules before any grouping is applied). The total of column *Rules* is not 19 because of duplicate rules. For instance, let us consider that a row lists 2 rules with system calls *A* and *B* on the LHS; and another row lists 3 rules, for system calls *A*, *C*, and *D* on the LHS. The total is thus 4 rules, one for each of *A*, *B*, *C*, and *D* on the LHS.

There are 16 rules generated correctly from at least one pair of strace logs (column *Correct*). In two cases, the algorithm generates rules that have too many system calls, as described in §4 (column *Partial*). For instance, when saving a file, VIM always issues system call `utime` (to get the file modification times) before `setxattrs` (to set the file attributes). As a result, the synthesized rule for `setxattrs` is always (incorrectly) prefixed with system call `utime`.

The incorrect rules were all due to misalignment, as explained in §4. The algorithm generated incorrect rules only for interactive programs due to their inherent non-determinism which affects our collection of system call traces: OpenSSH relies on random data, from both client and server; and VIM blinks the cursor a different number of times between executions.

Overall, these results are encouraging: The algorithm was able to generate most of the rules that it is designed to synthesize, simplifying the manual effort required.

6 Limitations and Future Extensions

The DSL is already expressive enough to support all the different scenarios that we present in this document. In this section, we identify the main limitations that we believe will need to be addressed to apply the DSL to additional scenarios.

Greedy rule matching. Currently, rules are matched greedily. As a result, rules cannot share a prefix of system calls on the LHS. We plan to explore alternative semantics to support this case.

Distant system call matching. Rules that match system calls separated by a large number of uninteresting system calls are long and require the DSL to keep all these system calls in memory. We plan to extend the DSL to support this scenario better.

Multithreading. In multithreaded programs, our implementation for Varan uses a separate DFA per thread. However, all DFAs use the same set of rules. Future versions of the DSL could include ways to map rules to specific threads.

Composing rules. Code blocks in the DSL cannot be combined. We plan to explore combining DSL rules and blocks of code, as the following example shows:

```
R1::{ return fd == 1; } R2::{ return fd == 2; }
nothing => write(fd,_,_) R1 || R2
```

Synthesizing from multiple pairs of traces. The synthesis algorithm struggles with under-represented system calls. Applying the algorithm to several pairs of traces would result in a higher count of the rare system calls.

Synthesizing predicated rules. When the pattern for a system call changes, the synthesis algorithm generates wrong rules due to misalignment. The algorithm can be extended by assigning an integer measure of confidence to each rule, which increases as the rule matches more system calls. After a threshold, refining yields a new rule instead of updating the current one.

Synthesizing more rules. The DSL synthesis algorithm only generates rules involving the original system call on the RHS. This algorithm can be extended to consider unmatched portions of the logs surrounded by matched sequences. For instance, for the divergence between lines 61–65 in Figures 5a and 5b, the rule presented in §5.4 can be extracted by considering the surrounding matched `fstat` and `close` in the strace logs.

7 Related Work

In this paper, we propose a DSL approach to reconcile system call divergences in the context of the multi-version execution [8,12,13,18,19,22,29,34,35]. Mx [18] performs dynamic software updates by running different program revisions in different versions. However, Mx can only deploy revisions that issue the same exact sequence of system calls; it does not tolerate any divergences. Varan [19] provides limited support for tolerating system call divergences through BPF filters [23] that rewrite the sequence of system calls. Varan was thus able to deploy program revisions that Mx could not. However, the BPF filters support only a single system call on the left-hand side, and the filters are very difficult to write by someone not familiar with BPF. Tachyon [22] also supports rewriting the sequence of system calls between two processes. However, it does not keep state between invocations by design; and it is also limited to a single system call on the left-hand side. Both Varan and Tachyon do not support the reordering of several system calls on the left to several others on the right, as we do in Figure 3b. On the other hand, the DSL we propose vastly improves the support for reconciling divergences, allowing for sophisticated rules that trigger only under certain conditions, which keep state about the current divergence being handled, which provide better support for complex reordering of system calls, and which require less effort to write given the expressiveness of the DSL.

Each DSL rule is compiled to a DFA, which resembles how regular expressions are efficiently implemented [1]. In fact, a RHS rule without predicates or callbacks is a regular expression for an alphabet in which each symbol is a system call. However, adding predicates to accept each symbol conditionally and callbacks after each symbol is accepted is an important difference.

Andersen and Lawall [2] propose a DSL and an algorithm for specifying and inferring generic patches to C programs which capture the collateral evolution of library call sites when the API changes. Instead, our DSL and synthesis algorithm operates on system call traces, which present specific challenges and opportunities.

Bakken et al. [4] propose a DSL for describing how to combine votes of multiple versions into a single result. Our DSL focuses on matching, instead of combining, the two "votes" on the recorded and replayed sides.

Techniques for synthesizing regular expressions from examples are related to the technique we propose for generating DSL rules from strace logs. Approaches based on genetic algorithms designed in the context of text extraction [6,7,10] are useful for generalizing the observed behavior to unseen examples. However, they do not match our goal of generating rules that exactly match all observed divergences. Regular expressions and DFAs can be synthesized through techniques that require a set of positive and negative examples [3,9,21,26]. These approaches are not directly applicable because part of the challenge of synthesizing DSL rules is to identify the positive examples, and the concept of negative examples does not apply directly to strace logs.

Program synthesis techniques [5,15,16,17,27] are also related to the technique we propose for generating DSL rules. The most relevant technique is λ^2 [15], which performs an enumerative search in a loop that generates candidates and refines them iteratively. Our technique generates a single candidate per rule using the first match and then refines it iteratively using the rest of the strace log. λ^2 can generate more than one candidate, and it uses a cost model to guide the program generation to generate a minimal cost solution. The extension we propose of adding a metric for the confidence of a candidate and keeping candidates above a threshold is similar.

8 Conclusion

In this paper, we have presented a simple and expressive domain-specific language (DSL) to write rules to tolerate expected divergences in the sequences of system calls issued by different program executions. The DSL we propose enables the deployment of multi-version execution (MVE) systems in a wider range of scenarios. In particular, we showed its applicability to three scenarios: (1) running versions of the same program under different configurations; (2) running different software releases, and (3) running native programs together with versions instrumented for dynamic analysis.

We report the results of an experimental evaluation for all the scenarios by manually writing the required rules starting from pairs of system call trace logs, obtained for each version in isolation. In particular, we show that the user needs no knowledge about the internals of the programs and analyses being deployed through MVE. We provide empirical evidence of the low effort required to identify and write such DSL rules, and we present the design and evaluation of an algorithm to automatically extract some of the DSL rules from such pairs of logs.

We believe that the DSL we propose, with its ability to easily encode divergences in the sequences of system calls issued by two executions, is an important contribution that will enable exciting new research on multi-version execution.

9 Acknowledgments

We thank our shepherd Gilles Muller, the anonymous reviewers, and Andrea Mattavelli and for their useful feedback on the paper. We also thank the EPSRC for sponsoring this research through an Early-Career Fellowship and a CDT studentship.

References

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison Wesley, 2006.

[2] Andersen, J., and Lawall, J. L. Generic patch inference. In *Proc. of the 23rd IEEE International Conference on Automated Software Engineering (ASE'08)* (Sept. 2008).

[3] ANGLUIN, D. Learning regular sets from queries and counterexamples. *Information and Computation 75*, 2 (Nov. 1987), 87–106.

[4] BAKKEN, D. E., ZHAN, Z., JONES, C. C., AND KARR, D. A. Middleware support for voting and data fusion. In *Proc. of the 2001 International Conference on Dependable Systems and Networks (DSN'01)* (July 2001).

[5] BAROWY, D. W., GULWANI, S., HART, T., AND ZORN, B. Flashrelate: Extracting relational data from semi-structured spreadsheets using examples. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'15)* (June 2015).

[6] BARTOLI, A., DAVANZO, G., DE LORENZO, A., MEDVET, E., AND SORIO, E. Automatic synthesis of regular expressions from examples. *Computer 47*, 12 (Dec. 2014), 72–80.

[7] BARTOLI, A., DE LORENZO, A., MEDVET, E., AND TARLAO, F. Inference of regular expressions for text extraction from examples. *IEEE Transactions on Knowledge and Data Engineering 28*, 5 (May 2016), 1217–1230.

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

[9] BONGARD, J., AND LIPSON, H. Active coevolutionary learning of deterministic finite automata. *Journal of Machine Learning Research 6* (Dec. 2005), 1651–1678.

[10] BRAUER, F., RIEGER, R., MOCAN, A., AND BARCZYNSKI, W. M. Enabling information extraction by inference of regular expressions from sample entities. In *Proc. of the 22th ACM International Conference on Information and Knowledge Management (CIKM'11)* (Oct. 2011).

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

[12] CHEN, L., AND AVIZIENIS, A. 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).

[13] COX, B., EVANS, D., FILIPI, A., ROWANHILL, J., HU, W., DAVIDSON, J., KNIGHT, J., NGUYEN-TUONG, A., AND HISER, J. 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] CRAMERI, O., KNEZEVIC, N., KOSTIC, D., BIANCHINI, R., AND ZWAENEPOEL, W. Staged deployment in Mirage, an integrated software upgrade testing and distribution system. In *Proc. of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)* (Oct. 2007).

[15] FESER, J. K., CHAUDHURI, S., AND DILLIG, I. Synthesizing data structure transformations from input-output examples. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'15)* (June 2015).

[16] GULWANI, S. Dimensions in program synthesis. In *Proc. of the 12th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'10)* (July 2010).

[17] GULWANI, S. Automating string processing in spreadsheets using input-output examples. In *Proc. of the 38th ACM Symposium on the Principles of Programming Languages (POPL'11)* (Jan. 2011).

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

[19] HOSEK, P., AND CADAR, C. Varan the Unbelievable: An efficient N-version execution framework. In *Proc. of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)* (Mar. 2015).

[20] KONING, K., BOS, H., AND GIUFFRIDA, C. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *Proc. of the 2016 46th International Conference on Dependable Systems and Networks (DSN'16)* (June 2016).

[21] LEE, M., SO, S., AND OH, H. Synthesizing regular expressions from examples for introductory automata assignments. In *Proc. of the 2016 International Conference on Generative Programming and Component Engineering (GPCE'16)* (Oct. 2016).

[22] MAURER, M., AND BRUMLEY, D. TACHYON: Tandem execution for efficient live patch testing.

In *Proc. of the 21st USENIX Security Symposium (USENIX Security'12)* (Aug. 2012).

[23] McCanne, S., and Jacobson, V. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. of the 1993 Winter USENIX Conference (USENIX Winter'93)* (Jan. 1993).

[24] Nethercote, N., and Seward, J. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science 89*, 2 (2003).

[25] Nethercote, N., and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'07)* (June 2007).

[26] Parekh, R., and Honavar, V. G. Learning dfa from simple examples. *Machine Learning 44*, 1-2 (July 2001), 9–35.

[27] Perelman, D., Gulwani, S., Grossman, D., and Provost, P. Test-driven synthesis. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'14)* (June 2014).

[28] Pina, L., and Hicks, M. Tedsuto: a general framework for testing dynamic software updates. In *Proc. of the IEEE International Conference on Software Testing, Verification, and Validation (ICST'16)* (Apr. 2016).

[29] Salamat, B., Jackson, T., Gal, A., and Franz, M. 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).

[30] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. AddressSanitizer: A fast address sanity checker. In *Proc. of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)* (June 2012).

[31] Serebryany, K., and Iskhodzhanov, T. ThreadSanitizer—data race detection in practice. In *Workshop on Binary Instrumentation and Applications* (12 2009).

[32] Seward, J., and Nethercote, N. Using Valgrind to detect undefined value errors with bit-precision. In *Proc. of the 2005 USENIX Annual Technical Conference (USENIX ATC'05)* (Apr. 2005).

[33] Stepanov, E., and Serebryany, K. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proc. of the International Symposium on Code Generation and Optimization (CGO'15)* (Feb. 2015).

[34] Volckaert, S., Coppens, B., Voulimeneas, A., Homescu, A., Larsen, P., Sutter, B. D., and Franz, M. Secure and efficient application monitoring and replication. In *Proc. of the 2016 USENIX Annual Technical Conference (USENIX ATC'16)* (June 2016).

[35] Xue, H., Dautenhahn, N., and King, S. T. 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).

[36] Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., and Bairavasundaram, L. How do fixes become bugs? In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'11)* (Sept. 2011).