FreeDA: Deploying Incompatible Stock Dynamic Analyses in Production via Multi-Version Execution

Luís Pina
George Mason University*
lpina2@gmu.edu

Anastasios Andronidis
Imperial College London
a.andronidis15@imperial.ac.uk

Cristian Cadar
Imperial College London
c.cadar@imperial.ac.uk

ABSTRACT

Dynamic analyses such as those implemented by compiler sanitizers and Valgrind are effective at finding and diagnosing challenging bugs and security vulnerabilities. However, most analyses cannot be combined on the same program execution, and they incur a high overhead, which typically prevents them from being used in production.

This paper addresses the ambitious goal of running concurrently multiple incompatible stock dynamic analysis tools in production, without requiring any modifications to the tools themselves or adding significant runtime overhead to the deployed system. This is accomplished using multi-version execution, in which the dynamic analyses are run concurrently with the native version, all on the same program execution.

We implement our approach in a system called *FreeDA* and show that it is applicable to several common scenarios, involving network servers and interactive applications. In particular, we show how incompatible stock dynamic analyses implemented by Clang's sanitizers and Valgrind can be used to check high-performance servers such as Memcached, Nginx and Redis, and interactive applications such as Git, HTop and OpenSSH.

CCS CONCEPTS

• **Computer systems organization** → *Reliability*; • **Software and its engineering** → **Runtime environments**; *Software testing and debugging*;

KEYWORDS

Multi-version execution, sanitizers, Valgrind

ACM Reference Format:
Luís Pina, Anastasios Andronidis, and Cristian Cadar. 2018. *FreeDA*: Deploying Incompatible Stock Dynamic Analyses in Production via Multi-Version Execution. In *Proceedings of CF '18: Computing Frontiers Conference, Ischia, Italy, May 8–10, 2018 (CF '18),* 10 pages.
https://doi.org/10.1145/3203217.3203237

* Most work done while the author was at Imperial College London.

CF '18, May 8–10, 2018, Ischia, Italy

ACM ISBN 978-1-4503-5761-6/18/05…$15.00
https://doi.org/10.1145/3203217.3203237

1 INTRODUCTION

Modern dynamic analysis tools, such as compiler sanitizers [38, 39, 41] and Valgrind [32] can find many different types of important errors without false positives. Unfortunately, such tools have two critical shortcomings. First, they are not modular and thus cannot be combined over the same program execution. For instance, with compiler sanitizers, users can choose to analyze their program execution for memory or address safety, but not both. Second, they introduce a high runtime overhead—for instance, Clang's memory and address sanitizers impose overheads of 2–4x, and Valgrind imposes 10–100x.

These two shortcomings generally prevent these tools from being applied in production, instead restricting their usage to offline testing. While offline testing is a key component of good software development, test suites exercise a limited, often artificial, set of executions. Therefore, the ability to complement offline testing with online in-production testing is important. In particular, tools such as compiler sanitizers and Valgrind could flag errors on executions that actually occur in production, providing detailed information necessary for debugging and fixing any errors found. Note that many of these errors could otherwise go unnoticed, silently corrupting the application state.

Our vision is to make popular stock dynamic analysis (DA) tools, such as compiler sanitizers and Valgrind, run as black boxes that can be seamlessly combined with low-enough overhead to be deployable in production. In this paper, we show that this vision is achievable in many practical scenarios, and we present *FreeDA*, a system which accomplishes this goal in a transparent way. More concretely, we show that *FreeDA* can: (1) deploy multiple analyses concurrently, even if such analyses are incompatible with one another, and (2) mask the runtime overhead of such analyses for typical scenarios involving high-performance servers and interactive applications.

FreeDA leverages *Multi-Version Execution (MVE)* [17, 19, 20, 23, 27, 29, 31, 37, 44, 46, 47] to run the native application concurrently with several DA versions (e.g., in parallel with Valgrind and compiler-sanitized versions), each in their own process. *FreeDA* leverages the record-replay strategy used in the second-generation of multi-version execution systems [28, 29, 44, 46], by recording the results of system calls issued by the native version into an in-memory *system-call buffer*, while each DA version reads the results of its system calls from this buffer. This separation allows *FreeDA* to run the native version at full speed (minus the small time needed to write the results of system calls into a buffer), while each DA version operates at a lower speed in the background.

However, given that the slower DA versions consume system calls at a much lower rate than the fast native version issues them, the system-call buffer can get full quickly, making this approach

impractical. Our work provides solutions that enable *FreeDA* to sustain native performance for indefinite periods of time for two common application scenarios: interactive applications (§2.2.1) and load-balanced server applications (§2.2.2). The basic idea is to exploit idle times between interactions/requests to allow the slow analyses to catch up with the fast native version. For server applications, *FreeDA* can also provide probabilistic error detection without an increase in latency through weighted load balancing. We also present an analytical model to estimate the minimum size of the system-call buffer for each case (§3).

As an additional challenge, given that *FreeDA* synchronises all versions at the system-call level, the sequences of system calls on all the versions must match. However, dynamic analyses can modify the system call sequence that the analysed program issues, resulting in divergences. We provide a taxonomy of the changes introduced by popular stock dynamic analyses and take advantage of prior work on multi-version execution to reconcile these changes (§2.1).

We provide a prototype implementation of these ideas and thoroughly evaluate its effectiveness on a variety of interactive and server applications using the popular stock dynamic analyses provided by ASan, MSan, TSan and Valgrind (§5).

In summary, this paper makes the following contributions:

(1) A low overhead multi-version execution system, *FreeDA*, that allows the deployment of popular incompatible stock dynamic analysis tools in production environments.
(2) A taxonomy of the changes introduced by popular stock dynamic analyses (ASan, TSan, etc.) that result into different sequences of system calls from the native application.
(3) An analytical model that predicts how many system calls need to be buffered to retain native performance indefinitely based on measurable properties of the native application.
(4) Two common scenarios in which *FreeDA* is immediately applicable: (1) interactive applications with enough idle times to absorb the overhead of DA checks, and (2) server applications in which idle time is introduced by splitting traffic through load balancing.
(5) An approach for checking a fraction of the requests sent to a network server in order to balance the tradeoff between checking coverage and resource consumption.
(6) A thorough evaluation involving the popular interactive applications Git, OpenSSH, HTop and Vim, and high-performance servers Memcached, Nginx and Redis, deployed together with the stock dynamic analyses implemented by Clang's sanitizers (ASan, MSan and TSan), and the Valgrind tool.

2 DESIGN

The architecture of *FreeDA*, shown in Figure 1, is similar to state-of-the-art second-generation MVE systems [28, 29, 44]. *FreeDA* executes several versions in the same MVE deployment, each in its own process: a fast *native version*, with which the users interact directly; and a *DA version* for each dynamic analysis.

FreeDA rewrites the binary code of each version, as they are loaded in memory, to intercept all system calls. For each system call that the native version issues, *FreeDA* forwards it to the operating system kernel, and copies the system call number, its arguments, and the return value to a *system-call ring buffer* (*SCB* for short)

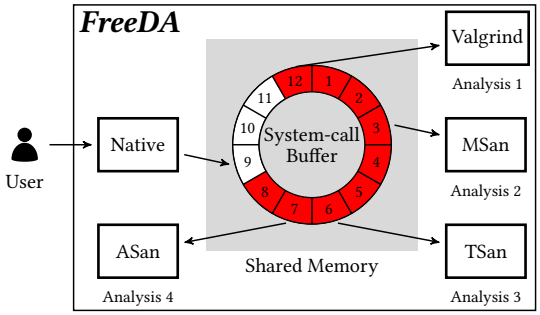

Figure 1: Architecture of *FreeDA*. Red entries are full.

located in memory shared among all the versions. For each system call that each DA version issues, *FreeDA* matches it with the next system call on the *SCB* and takes the results that the native version registered.

The decentralized architecture of *FreeDA* has two main advantages. First, *FreeDA* enables the deployment of multiple incompatible stock dynamic analyses *over the same program execution*, as each analysis is executed independently on a dedicated process and all sources of input/output and non-determinism are intercepted by *FreeDA*. In the example shown in Figure 1, there are four such DA versions: one where the application runs under Valgrind's memcheck (*Valgrind*), one compiled with memory sanitization (*MSan*), one with thread sanitization (*TSan*), and one with address sanitization (*ASan*).

Second, *FreeDA* is well suited for deploying slow analyses trailing behind a fast native version. *FreeDA* executes the fast native version of the application at full speed, minus the small overhead of registering system calls in the shared *SCB*, thus providing a fast response time to the user. Each analysis then makes progress at its own pace, in its own separate (background) process. Figure 1 shows a native version running ahead of all analyses, currently registering a new system call in position 9. Each analysis executes at its own speed and consumes a different position: Valgrind consumes position 12, while MSan consumes position 3, and so on.

However, *FreeDA* faces two challenges: (1) The dynamic analyses may change the sequence of system calls issued during execution, and (2) the buffer may become full because the analyses are significantly slower than the native version. Note that when the latter happens, the native version will execute at the speed of the slowest analysis, as each attempt to write into the buffer causes the native version to block until the slowest analysis frees a position.

FreeDA overcomes both challenges. First, it takes advantage of prior work [28, 31, 36] to reconcile divergences in the sequences of system calls between the native version and each analysis using rewrite rules (§2.1). Second, to prevent the *SCB* from becoming full, *FreeDA* exploits idle times in interactive applications due to slow user input, and introduces additional idle time to server applications by splitting traffic through load balancing (§2.2).

2.1 System-call changes introduced by DA tools

To run stock dynamic analyses, *FreeDA* must reconcile the system-call changes they introduce. A key observation is that these changes fit into a small number of categories:

(1) **Intercepting.** Dynamic analyses intercept some system calls that the program under analysis issues and transforms them using a set of fixed patterns. The same pattern often applies to many different system calls:
   (a) **Wrapping.** A system call issued by the program is surrounded by additional system calls before and/or after it. For instance, Valgrind wraps most system calls with two extra system calls: one to disable signal delivery just before the original system call, and another to re-enable it immediately after. Or, ASan adds a `sched_getaffinity` after a `fork`.
   (b) **Modifying.** A sequence of system calls is replaced by a different sequence. For instance, Valgrind transforms an `open` system call into a sequence of `dup`, `lseek`.
   (c) **Skipping.** A system call issued by the program is skipped completely. For instance, most analyses monitor some signals (e.g., `SIGSEGV`) by installing signal handlers. If the program under analysis registers its own handler, the analysis just updates its internal state.

(2) **Weaving.** The analysis has additional functionality which does not exist in the target program. The system calls invoked by this additional functionality do not result from intercepting any system call issued by the target program. There are fixed regular patterns that analyses use:
   (a) **Initialization/Teardown.** Analyses usually introduce extra system calls in the beginning and end of the program execution, to initialize their internal state, perform sanity checks, or gather statistics after the program ends.
   (b) **Error reporting.** When the analysis finds an error, it writes a message to a file or to standard output.
   (c) **Bookkeeping.** System calls that manage the internal state of the analysis. For instance, when Valgrind JITs code, it has to manage the code cache and sometimes allocate more memory through system call `mmap`.

As an example, consider an application performing the following sequence of three system calls:

```
1 open("/proc/self/cmdline", ...)=4
2 read(4,...,4096)=188
3 rt_sigaction(SEGV,0x7f932...,0,8)=0
```

Valgrind transforms this sequence as follows:

```
1.1 dup(1025)=4
1.2 lseek(4,0,SEEK_SET)=0
2.1 gettid()=29387
2.2 write(1029,"Z",1)=1
2.3 rt_sigprocmask(SIG_SETMASK,[],~[ILL,TRAP,...],8)=0
2.4 read(4,...,4096)=188
2.5 rt_sigprocmask(SIG_SETMASK,~[ILL,TRAP,...],0,8)=0
2.6 gettid()=29387
2.7 read(1028,"Z",1)=1
```

The sequence starts by opening a file that contains the command line used to launch the current process. Earlier in execution, during its initialization, Valgrind creates a temporary file with the original command line (i.e. without Valgrind and its arguments), and keeps it open on file descriptor 1025. During execution, Valgrind replaces the original `open` system call, on line 1, with a `dup` and `lseek`, on lines 1.1–1.2, to provide the program with a file descriptor with the original command line.

Valgrind *wraps* the original `read` system call with 6 system calls, between lines 2.1 and 2.7. In the first and last two system calls, Valgrind synchronizes itself internally by writing to and reading from file descriptors 1029 and 1028, respectively. Then, Valgrind disables signal delivery on line 2.3, issues the original system call on line 2.4, and re-enables signal delivery on line 2.5. Valgrind wraps in this way most system calls that the program under analysis issues.

In the original program, system call `rt_sigaction` installs a signal handler for signal `SIGSEGV`. This is an example of a skipped system call: Valgrind installs a handler for signal `SIGSEGV` during initialization, to detect memory errors, and handles system call `rt_sigaction` by simply updating its internal state.

FreeDA takes advantage of prior work on multi-version execution to reconcile these changes through a small number of rewrite rules [28, 31, 36]. We refer the reader to prior work, particularly [36], for details on how these rules are implemented in a multi-version execution context. In total, only 3, 1, 4, and 14 rewrite rules are needed to reconcile the changes introduced by ASan, MSan, TSan and Valgrind, respectively.

2.2 Sustaining native performance

Existing research on multi-version execution focuses on scenarios in which all the versions executed in parallel have the same performance characteristics (e.g., two program versions with stacks running in opposite directions [37] or two releases of the same application [27]). *FreeDA* is the first multi-version execution system that faces the challenge of running versions with widely different performance characteristics. In particular, because the native version is significantly faster, it will eventually fill the buffer, decreasing its speed to that of the slowest analysis.

In this paper, we focus on two practical scenarios in which *FreeDA* overcomes this challenge: interactive (§2.2.1) and server applications (§2.2.2).

2.2.1 Interactive applications. Interactive applications are structured around a loop that waits for input from the user and then processes it. For much of their execution time, interactive applications wait for user input. This is a key observation: Given a large enough buffer, *FreeDA* can mask the latency of a slow dynamic analysis by allowing the user to interact with the fast native version, while executing each analysis in the background. During the idle times the native version waits for user input, the analyses will continue to execute and eventually catch up.

The *SCB* must be large enough to accommodate the peak of program activity that follows each user input. Depending on the particular pattern of interaction and the slowdown that the analysis introduces, the *SCB* may need to be larger to accommodate more than one user input. However, as we shall see in §5, an *SCB* of 1Mi (2^{20}) entries (64MiB) is enough to mask the latency of techniques as heavyweight as Valgrind.

It is interesting to note that a subtle phenomenon allows *FreeDA* to slightly speed up the execution of the DA versions: When each DA version issues a system call, the results are *immediately ready* in the *SCB*. This means that the DA version effectively saves the time that it would otherwise spend inside the OS kernel.

2.2.2 Server applications. High-performance network servers often have both demanding throughput and latency requirements, and under load, there are no pause times that *FreeDA* can use to

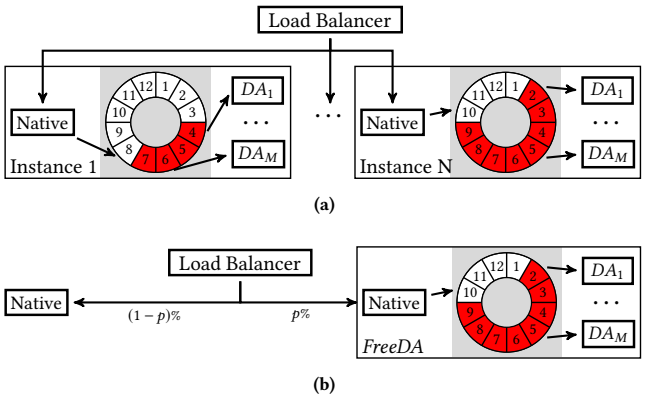

Figure 2: Deploying M different dynamic analyses on server applications with a load balancer that checks all requests using N instances of *FreeDA* (2a), or checks $p\%$ requests using a *FreeDA* instance and sends the remaining $(1-p)\%$ requests to a native instance (2b).

empty the *SCB*. Our novel strategy for overcoming this challenge is the new idea of combining a multi-version execution framework with a load balancer in order to artificially introduce idle times.

The high-level architecture of our deployment is depicted in Figure 2a, which shows a load balancer managing N different *FreeDA* instances. Each instance runs a native application (the one interacting with the load balancer) and M different DA versions. In this deployment, each instance processes a fraction of the requests. Between requests, each instance has time for the slower DA versions to catch up while other instances are processing requests. To make things concrete, if a native version would support up to 1000 request per second, then, by splitting the traffic into two, we introduce 50% idle time in each instance, which the DA versions can exploit.

We also propose another novel approach to combine load balancing with multi-version execution: Use a *weighted load balancer* to check $p\%$ requests with a *FreeDA* instance, and send the remaining $(1-p)\%$ to a native instance. This deployment, depicted in Figure 2b, provides a convenient trade-off between the percentage of the execution analyzed for errors and the hardware resources utilized, which allows to deploy heavier analyses using less hardware while retaining the native performance at the cost of not analyzing a portion of the execution. Of course, this partial checking deployment can scale up in both number of native instances, for increased throughput, or number of *FreeDA* instances, for increased checking coverage ($p\%$).

Importantly, we note that employing a load balancer to deploy the analyses without using *FreeDA* is not enough. Load balancing can increase the throughput when additional instances are added, but cannot improve latency. In contrast, MVE reduces the maximum throughput to restore latency close to the native level by replying to each request as soon as the fast native version finishes. It is only by this novel combination that *FreeDA* achieves *both* high throughput *and* low latency.

3 ANALYTICAL MODEL

In this section, we present an analytical model of *FreeDA*'s behavior with respect to the required space in its *SCB*. For simplicity, we consider the case of running a single analysis in the background. This model is valid for multiple analyses by simply reserving the maximum buffer size required for each analysis.

When a request[1] arrives to the native version, *FreeDA* starts storing system calls in the *SCB*. Ideally, if every new request comes exactly after the analysis has finished processing the previous request, then *FreeDA* only needs to store the system calls of one request. In general, *FreeDA* requires additional capacity to store new requests that arrive while processing an earlier request. Of course, if every request always arrives before the analysis has finished earlier requests, any finite buffer will eventually become full.

A load L (requests/time) greater than the throughput of the analysis TP_a fills a buffer of size B in time t_{full}. Assuming that each request issues S_{req} system calls, an upper bound of t_{full} is:

$$t_{full} = B/((L - TP_a) \times S_{req}) \quad (1)$$

For example, a load of $L = 11$ req/s applied to *FreeDA* deploying an analysis with throughput $TP_a = 10$ reqs/s, with each request issuing $S_{req} = 10$ system calls, fills a buffer of size $B = 100$ in at most $t_{full} = 10$s. After t_{full}, *FreeDA* performs with the latency of the analysis.

Of course, if $L \leq TP_a$, then the analysis always consumes all requests in every time unit. *FreeDA* can execute with native performance if $L \leq TP_a$ over a time period T, even if $L > TP_a$ during smaller intervals of T. For instance, consider a server that receives a request every 100ms, which the native version processes in 20ms but the analysis takes 70ms. Then, $L \leq TP_a$ holds over $T = 100$ms but not $T = 50$ms. *FreeDA* needs to store all N_T requests issued during T, which yields a buffer of size: $B = N_T \times S_{req}$. Choosing the time scale of T thus has a direct impact on the size of the buffer. Large units (e.g., hours) will result into a large buffer, while small units might violate $L \leq TP_a$.

We can improve this bound by observing that when the native version issues a certain number of requests, the analysis will have consumed a portion of them, proportional to its performance speed. For instance, if *FreeDA* is deployed behind a load balancer that guarantees 12 req/s, it should provision a buffer of size $B = 12 \times S_{req}$. However, let us also consider that the analysis *FreeDA* deploys is $F = 3$ times slower. When the native version finishes all 12 requests, the analysis will have consumed around $N_T/F = 4$ of them. Thus, instead of a buffer of size 12 requests, we now require 8+1 (we need an extra unit to count the currently consumed one). Given that a particular implementation of *FreeDA* may require several buffer entries N_{sys} per system call, the space required is thus:

$$B = (N_T \times S_{req} - N_T/F \times S_{req} + S_{req}) \times N_{sys} \quad (2)$$

Our approach ensures that $L \leq TP_a$ for a time period T by either adding additional capacity for network servers, or exploiting idle times for interactive applications. For servers, well-tuned load balancers can provide guarantees on the time between requests and thus ensure a small enough B. For interactive applications we have two cases. Short-lived interactive applications have a single request, so B should be simply S_{req}. For long-lived interactive applications,

[1] In this section we use network server terminology, but the model holds for interactive applications too, by replacing *request* with *interaction*, *latency* with *response time*, etc.

we can choose appropriate T and N_T by understanding every how often the user is idle, e.g., measuring how fast average users type and how much time they spend reading the results on the screen.

4 IMPLEMENTATION

FreeDA shares similarities with modern MVEs based on a decentralized ring buffer architecture, such as Varan [28], MvArmor [29] and ReMon [44]. We decided to implement a prototype of *FreeDA* on a thorough re-engineering of Varan because it is maintained in our group and we find it simpler for this purpose than ReMon (which includes a kernel module) and MvArmor (which requires a virtualized environment). In Varan's terminology, *FreeDA* runs the native version as the *leader* and each analysis as a *follower*.

Non-determinism and multi-threading. Varan already handles non-deterministic and multi-threaded applications [28]. Varan handles naturally all sources of non-determinism that involve system calls (e.g., reading data from `/dev/random`), since the followers just read the results of system calls from the *SCB*. *FreeDA* thus supports OpenSSH, which uses such random data (§5.1). Varan captures the order in which threads issue system calls in the leader and enforces that same order on every follower. *FreeDA* thus supports multi-threaded applications that synchronize threads through system calls (e.g., Memcached described in §5.2).

This approach, however, fails for programs that synchronize threads using memory operations, e.g, Git (see §5.1). For instance, pthreads implement mutual exclusion (*mutexes*) through a *compare-and-swap* operation. *FreeDA* extends Varan to support pthreads by: extending Varan to intercept calls for locking and unlocking mutexes, registering them on the *SCB* on the native leader, and enforcing that each DA version obtains the same mutexes in the same order.

We show in §5 that *FreeDA* supports running multi-threaded applications under analyses such as ASan, MSan, and TSan. However, it does not support analyses that use their own thread scheduling algorithm, such as Valgrind. In this case, the ordering that *FreeDA* enforces may clash with the one that the analyses impose, which results in followers deadlocking. One possibility is to change the scheduler of such DAs to support preempting system calls (e.g., reschedule when a system call returns error code `EAGAIN`).

Synchronizing file descriptors. Inherited from Varan, *FreeDA* uses an extra process—the *monitor*—to propagate open file descriptors from the native version to all followers through UNIX sockets. The monitor is needed because the native version does not know the identity nor the number, if any, of DA versions running in the background. The monitor is a single-threaded loop that: (1) waits for a file descriptor from the leader and (2) sends the file descriptor to each follower, in sequence. Due to the nature of UNIX sockets, all steps require synchronous communication. This is not a problem when all versions run at similar speeds, but slows down the fast leader when it needs to synchronize with a slow follower. *FreeDA* addresses this problem by using a producer-consumer monitor with a thread to receive file descriptors from the leader and store them in a *file-descriptor buffer*, and another thread to send each file descriptor to all analyses.

5 EVALUATION

This section describes our empirical evaluation of *FreeDA*. We show that *FreeDA* can successfully deploy popular stock dynamic analyses for two types of common applications: interactive programs (§5.1) and network servers (§5.2).

We used four stock dynamic analyses for C/C++ programs in our experiments: *Valgrind's memcheck* version 3.11 (revision 15920, VEX revision 3233), and the compiler sanitizers *ASan*, *MSan*, and *TSan* that ship with Clang version 3.8. *Valgrind's memcheck* is a tool that checks for uses of invalid memory (i.e. uninitialized, unallocated, or freed memory) through heavyweight dynamic instrumentation [40]. *ASan* is the *address compiler sanitizer* [38] that detects buffer overflows and use-after-free errors. *MSan* is the *memory compiler sanitizer* [41] that detects uses of uninitialized memory. *TSan* is the *thread sanitizer* [39] that detects data races. Valgrind, ASan, and TSan are readily applicable to existing programs. MSan requires some application changes (e.g., marking data allocated inside libraries as initialized), thus we only used it for one application (§5.1) after the appropriate code changes.

We refer to buffer sizes in terms of their capacity as follows: A buffer that holds 1024 entries has capacity 1Ki entries and uses 64KiB of memory (64 bytes per entry). The maximum buffer used has a size of 1Mi and consumes 64MiB of memory. Note that our implementation of *FreeDA* requires buffer sizes to be powers of two and uses two buffer entries per system call (entry and exit events, $N_{sys} = 2$ from §3).

Each bar reported in our graphs is the average of five measurements; error bars show the standard deviation.

5.1 Interactive Programs

We used two types of interactive applications in our evaluation: short-lived command-line utilities (Git and OpenSSH) and text-based long-lived applications (HTop and Vim). In both cases, we are concerned with the response time perceived by the user. For short-lived command-line utilities, the response time is simply the execution time of the utility. For long-lived applications, it is the period between the moment the user provides input to the application (e.g., enters a string to be searched in Vim and presses *Enter*) and the time the result is returned to the user. We automated such interactions using the *expect* tool [14].

We tested the interactive applications on a machine equipped with two 2.50GHz Intel Xeon E5-2450 v2 CPUs (8 physical cores, 16 logical per CPU), with 188G of RAM, and running 64-bit Ubuntu 16.04 (kernel version 4.4.0-45, glibc version 2.23). We now describe the experiments for each application.

Git is a widely-used version control system [2]. We benchmarked Git version 2.9.2 by using four common commands: (1) `log -stat HEAD~200`, which lists the 200 most recent commits and statistics about files changed in each commit, (2) `blame`, which shows the author that last modified each line of a file, (3) `diff`, which compares the changes between two versions, and (4) `tag`, which lists all the tags. We ran these commands on the *CMake* [1] repository, which had 32,374 modifications (commits) at the time of our experiments.

Figure 3a shows the time required to run each command. We evaluate five different deployments of Git: the native version of the

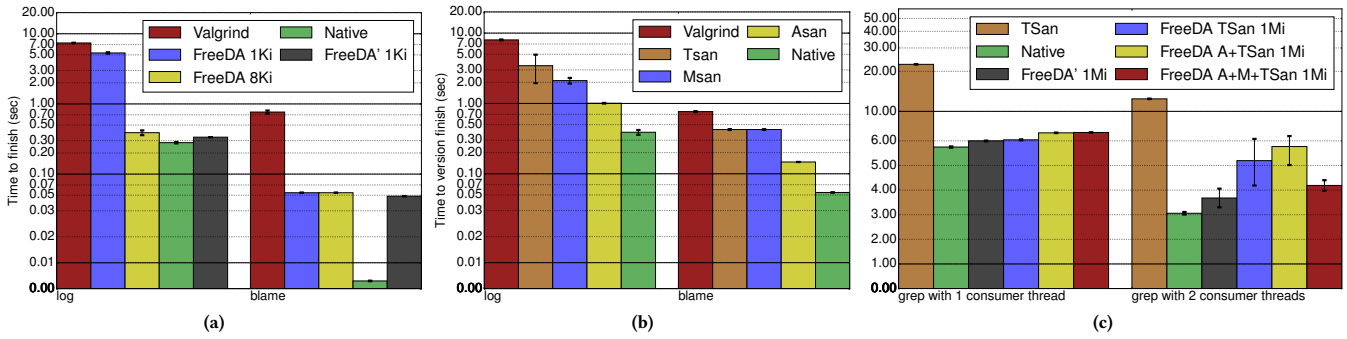

Figure 3: Time required to run the Git tool for different commands with and without *FreeDA* (3a); time required for each individual version under *FreeDA* to finish for each command, with an 8Ki *SCB* size (3b); and time required to run a concurrent Git command with one and two consumer threads (3c).

tool (*Native*), Valgrind without *FreeDA* (*Valgrind*), *FreeDA* with a native version running in the background instead of an analysis and using a 1Ki *SCB* (*FreeDA' 1Ki*), and *FreeDA* with all four analyses and two *SCB* sizes: 1Ki (*FreeDA 1Ki*) and 8Ki (*FreeDA 8Ki*) entries.

Valgrind adds significant overhead (note that the scale is logarithmic). For instance, the response time for *git log* increases from under 0.3s to over 7s (over 23x), and that for *git blame* from under 5ms to around 700ms (over 140x). *FreeDA' 1Ki* shows that the overhead that *FreeDA* itself introduces, by registering the system calls on the shared buffer, is low. Note that the startup time of *FreeDA* results in a large overhead in relative terms, but small in absolute terms (around 50ms).

FreeDA 8Ki has roughly the same performance as *FreeDA' 1Ki*, adding little overhead on top of the native execution. Also, note that all native execution times under 100ms (unnoticeable to the human eye) are kept under 100ms by *FreeDA 8Ki*. With the exception of one command, `git log`, the smaller buffer of *FreeDA 1Ki* is enough for these benchmarks. We measured that `git log` issues around 3200 system calls. From §3, we can predict a minimum buffer size of around 6400 entries for such short-lived applications which, given *FreeDA*'s requirement for power-of-two buffer sizes, means a buffer size of 8Ki. We confirmed that a buffer of size 4Ki behaves as one of 1Ki, and that a buffer of 32Ki does not improve the results over 8Ki. The results for `git log` with a buffer size of 1Ki show how a buffer smaller than necessary affects performance.

Figure 3b shows how long the native leader and the four DA versions took to execute inside *FreeDA*. Note that the *Native* bars in this figure are the same as the *FreeDA 8Ki* bars in Figure 3a: *FreeDA* ends execution and provides the user with a result as soon as the fast native version terminates. The slower analyses finish in the background without affecting the perceived latency. Comparing the *Valgrind* bars in Figure 3a (which show the time taken by Valgrind when run by itself) with the *Valgrind* bars in Figure 3b (which show the time taken by Valgrind when run inside *FreeDA*), we can see that *FreeDA* does not add any overhead to the DA versions. In fact, as we discussed in §2.2, the analyses run under *FreeDA* sometimes get *faster*, as they read the results of system calls directly from the *SCB*, without any context switching into the kernel or any I/O waiting.

We also ran the command `git grep .` which has a multithreaded implementation with one producer thread that generates work (files to be searched) and a configurable number of consumer threads. We ran this command with three analyses: ASan, MSan, and TSan Figure 3c shows the results, comparing with the slowest analysis used: TSan. As before, *FreeDA'* deploys a native version in the background instead of an analysis. For one thread, the extra events from intercepting `pthread` calls stress the *SCB* enough to show some expected overhead when increasing the number of followers. For two threads, the relative overhead is higher and the results are noisy because each version has access to fewer physical cores than active threads (2 cores per version). Even though *FreeDA* has non-negligible overhead when not enough cores are available, it still finishes execution much faster than ASan, MSan, or TSan alone, significantly reducing the latency that the user experiences (i.e. *FreeDA* shows the result to the user as soon as the native version produces it). Adding more analyses increases the overhead due to the underlying Varan implementation [28].

We repeated these experiments on two other repositories—namely *Memcached* [7] with 1,220 modifications and *Git* [3] with 43,856—and obtained similar results.

OpenSSH is a suite of utilities used to secure communication by encrypting network traffic [10]. We used version 7.1. OpenSSH uses the cryptographic primitives provided by the OpenSSL library [11]. We used OpenSSL 1.0.1 with two changes. First, we compiled OpenSSL with `-DPURIFY` to remove undefined behavior [12] which causes the DA versions to generate different random data and diverge. Second, we added an option to initialize the random number generator with a deterministic seed instead of using `/dev/random`, to generate the same keys across different executions of OpenSSH.

We benchmarked OpenSSH by (1) executing the program `true` (which simply exits with a zero return code) on the same machine through the command `ssh localhost`, and (2) generating authentication keys of size 4096 bits through command `ssh-keygen`, for three different random seeds. The seed determines the execution time, and we used three seeds that result in fast (~0.3s), medium

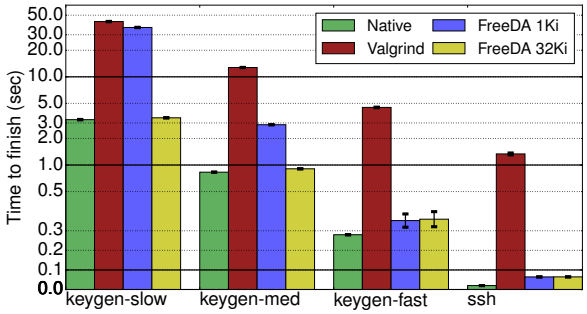

Figure 4: Time required to run the SSH command-line tools. The numbers in the legend refer to the size of the *SCB*.

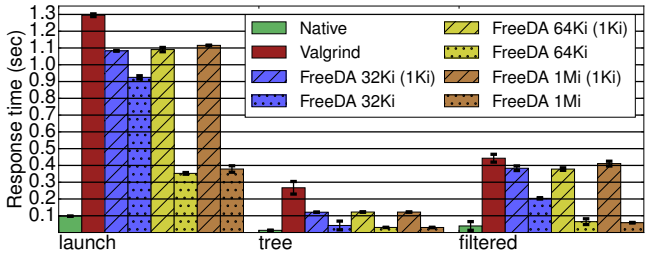

Figure 5: Response times for various tasks in HTop. The first number in the legend is the size of the *SCB*, and the second number is the size of the file-descriptor buffer.

(~0.7s) and slow (~3.0s) run time. Our results are shown in Figure 4. Even though Valgrind adds prohibitive overheads (up to 16x), *FreeDA* is able to deploy all three analyses with negligible overhead with a buffer size of 8Ki, predicted by inputing the number of system calls of each command in Equation 2 in §3.

HTop is an interactive system monitor and process viewer [5]. We designed a realistic interaction scenario between a user and HTop version 2.0.1: the user launches HTop (*launch*); turns on tree view (*tree*); presses the / key, types a pattern to filter, presses *Enter* and waits for the result (*filtered*). Our scripts simulate a user latency of one second between actions. HTop opens a large number of files during normal operation, so the file-descriptor optimization described in §4 was critical.

Figure 5 shows the response time for each simulated activity. Valgrind causes HTop to run noticeably slower between each user input. *launch* issues around 20Ki system calls, *tree* issues 6Ki, and *filtered* issues between 13Ki–18Ki. Considering an average of $S_{req} = 13Ki$ system calls for each $N_T = 1$ interaction, and a slowdown of $F = 13$ for Valgrind (measured by comparing the *native* and *Valgrind* bars in Figure 5), Equation 2 in §3 predicts an *SCB* size of 64Ki. Once again, our results confirm the prediction, given the poor performance for 32Ki and the lack of improvement for 1Mi over 64Ki. Increasing the size of the *SCB* only results in better performance up to a point, after which we need to increase the size of the file descriptor buffer. In the end, with large enough buffers, *FreeDA* is able to mask the overhead of all three analyses running in the background.

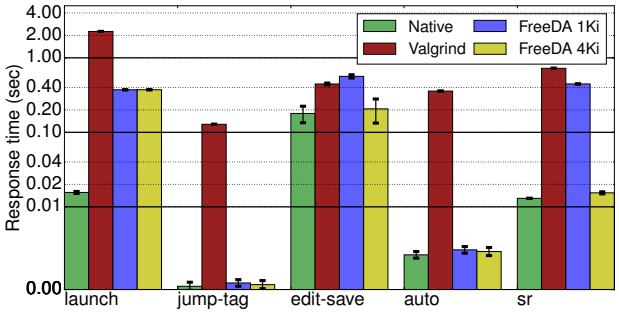

Figure 6: Response times for various tasks in Vim. The numbers in the legend refer to the size of the *SCB*.

Vim is a screen-oriented text editor. We designed a realistic scenario for a user editing a file containing C code: the user launches Vim (*launch*); jumps to a line by number, which has a function call, and jumps to that function definition through a tag (*jump-tag*); adds an extra argument to the function and saves the file (*edit-save*); renames the function using the auto-complete feature in Vim (*auto*); and, finally, performs a document-wide search-and-replace, saving the document afterwards (*s&r*). Our scripts wait for one second between each action to simulate user latency. We measured the number of system calls issued for each command, and used Equation 2 in §3 to predict a buffer size of 4Ki. Our results are shown in Figure 6. As in previous experiments, with a large enough buffer, the overhead of the analyses run in the background, particularly Valgrind, is masked by *FreeDA*.

5.2 Server Programs

We tested *FreeDA* with three high-performance widely-used network servers. **Nginx** [9] is a popular reverse proxy server, often used as an HTTP web server, load balancer, or cache. We benchmarked Nginx 1.11.2 with *wrk2* [16] 4.0.0. **Redis** [13] and **Memcached** [6] are high-performance in-memory key-value data stores, used by many well-known services. We benchmarked Redis 3.0.7 and Memcached 1.4.36 with *memtier* [8] 1.2.10. We also used load balancers HAProxy [4] 1.6.7 and Twemproxy [15] 0.4.1.

We conducted our experiments on a cluster of three machines (M1, M2, and M3), all located on the same rack and connected by a 1Gb Ethernet link. **M1** is the machine described in §5.1. **M2** and **M3** are two identical machines, each with a 3.50 GHz Intel Xeon E3-1280 CPU (4 physical cores, 8 logical) and 16 GB RAM running 64-bit Ubuntu 14.04 LTS (kernel version 3.13.0-88).

5.2.1 Direct connection. We tested Nginx with a single analysis, ASan, over a direct connection (i.e. no load balancing) with three deployments: native, ASan, and *FreeDA* with ASan. M2 runs the Nginx server configured to serve a 2KiB file, containing random data, with protocol-level compression enabled. M1 runs *wrk2*, transferring the served file for 50 seconds using 4 threads and 80 open connections.

We saturated the ASan deployment by running *wrk2* with increasing throughput, from 7K req/s, with a step of 1K req/s, until the server could not maintain the throughput. We also measured the latency reported for each throughput value.

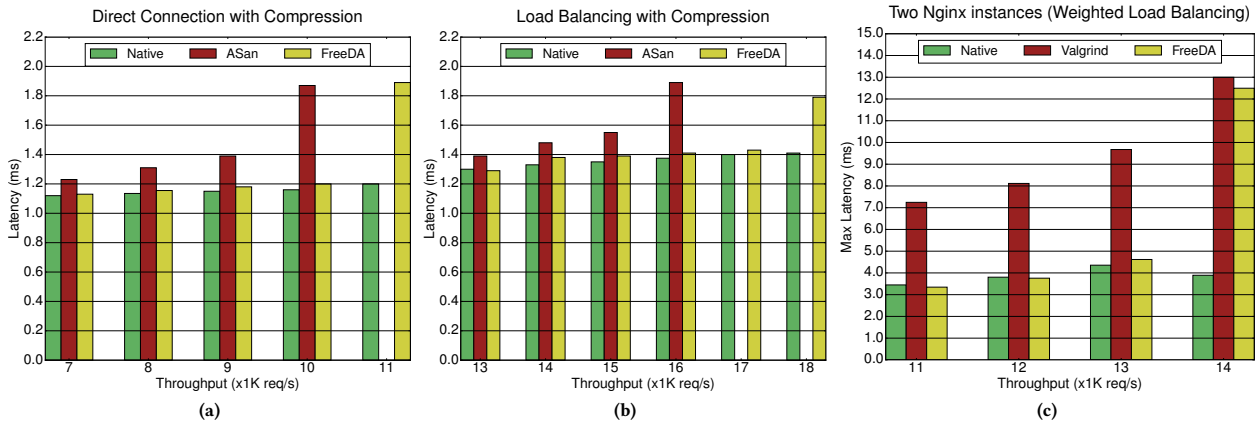

Figure 7: Results for the Nginx experiments: (a) Latency of Nginx deployed natively, with ASan, and with ASan through *FreeDA* via a direct connection, (b) same experiment with two Nginx instances behind a load balancer, and (c) same experiment redirecting 19/20 of the requests to a native instance and 1/20 to an instance deploying either Valgrind or Valgrind through *FreeDA*.

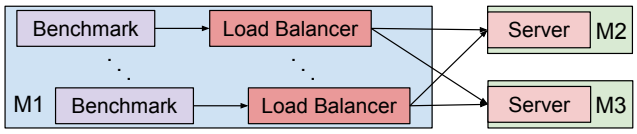

Figure 8: Architecture of the distributed experiments.

Figure 7a shows the results. ASan's maximum throughput is 10K req/s (the native's maximum throughput is 14K req/s). ASan increases the latency significantly, by 61.2% at 10K req/s, while *FreeDA* increases the latency by just 3.5% at 10K req/s (and even less for lower throughputs). The bar for ASan at 11K req/s is missing because ASan cannot sustain that throughput on this experiment.

We used a buffer size of 128, which we computed with Equation 2 in §3 as follows. Each Nginx request issues an average of $S_{req} = 15$ system calls. Given that the benchmark uses 4 threads, we should provision enough buffer space for $N_T = 4$. We can observe that ASan introduces a slowdown of $F = 1.5$ at worst (10K req/s in Figure 7a). We also ran this experiment with a buffer size of 256, and we observed no improvement over the results we report here.

5.2.2 Load balancing Nginx. We repeated the previous experiment, deploying Nginx behind a load balancer. M1 runs one pair of *wrk2* and the HAProxy load balancer, M2 and M3 run one instance of Nginx each. Figure 7b presents the results, which are similar to the previous experiment: *FreeDA* is able to hide the overhead of ASan, while maintaining a higher throughput than ASan. We do not observe a 2x increase in overall throughput as load balancing itself introduces overhead.

FreeDA achieves a higher throughput, 17K req/s, because the DA version reads the results of system calls directly from the shared buffer, saving kernel context switches, as explained in §2.2.2. However, past this throughput, the buffer becomes full and the latency of *FreeDA* increases to that of ASan. Note that there are bars missing for ASan at 17K req/s and 18K req/s because ASan cannot sustain these throughputs on this experiment.

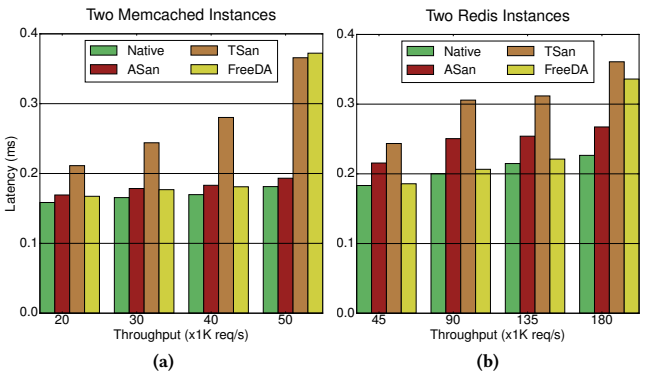

Figure 9: Two Memcached instances (9a) and two Redis instances (9b) behind a load balancer.

5.2.3 Load balancing Redis and Memcached. We repeated the previous experiment for Redis and Memcached. We used the setup shown in Figure 8, with M1 running Twemproxy and the *memtier* benchmark, and M2 and M3 running an instance of Redis or Memcached each. This time, Twemproxy became a bottleneck. We saturated the server with multiple pairs of *memtier*/Twemproxy on M1, each configured for 10K req/s for Memcached, and 45K req/s for Redis. Each *memtier* instance issues the same number of `GET` and `SET` operations with values of 100 bytes. We used a buffer size of 128 for Redis, predicted using Equation 2 in §3 with $N_T = 4$, $S_{req} = 17$, and $F = 1.5$. For Memcached, we used a buffer size of 256, which was predicted in the same way with: $N_T = 4$, $S_{req} = 25$, and $F = 2$.

For both Redis and Memcached, we deployed two analyses in the background, ASan and TSan. Figures 9a and 9b show the results for Memcached and Redis, respectively. We observe a similar behavior to the Nginx experiment, with *FreeDA* hiding ASan's and TSan's latency while also achieving a slightly higher throughput.

5.2.4 Weighted load balancing. Heavyweight analyses like Valgrind would require a large amount of server instances to scale. As discussed in §2.2, an alternative to avoid such a substantial resource increase is to *sample* the traffic by configuring the load balancer to redirect only a portion of the requests to a *FreeDA* instance. Of course, sampling traffic means that we may miss bugs, but this is a useful mechanism to control the trade-off between the analysed traffic and utilization cost.

In our experiment, we used *FreeDA* with Nginx and Valgrind. We measured that Valgrind maintains a maximum throughput of 1/20 of the native's, so we configured HAProxy to send 1/20 of the total traffic to a *FreeDA* instance with Valgrind, and the remaining 19/20 to a native Nginx instance. We then repeated the previous experiment, serving a 2KiB file of random data with protocol compression enabled.

Figure 7c shows the maximum latency reported by *wrk2*. Note that we report the maximum latency because the 1/20 requests reaching Valgrind do not influence the average latency visibly. As expected, Valgrind increases the maximum latency dramatically, while *FreeDA* results in near-native maximum latency up to a throughput of 13K req/s.

6 DISCUSSION

FreeDA pays a price in terms of utilization overhead. However, many cores are left idle, and could be used by *FreeDA*. Furthermore, our weighted load balancer approach for server applications allows one to trade off checking coverage for utilization overhead, and the decentralized architecture of *FreeDA* also allows to terminate the execution of DA versions when resources are needed for other tasks.

This paper focuses on deploying stock dynamic analyses in production, stopping the execution if any of the analyses detects an error. Still, *FreeDA*'s ability to deploy incompatible analyses on the same execution can be useful in a testing context too, by deploying several analyses on the same test run for the cost of the slowest analysis (assuming the buffer gets full during testing).

We also note that *FreeDA* does not change the accuracy of the analyses, is applicable beyond bug-finding dynamic analyses such as Valgrind and compiler sanitizers—e.g., it could be equally used to deploy other types of analyses, e.g., profilers, collecting detailed trace logs during production runs, and more.

7 RELATED WORK

Both the research community and industry have put a lot of effort into designing many types of dynamic analysis techniques [18, 26, 32, 38, 39, 41]. However, many of these analyses are mutually incompatible and cannot be run together on the same program execution.

RepFrame [25] enables running incompatible stock analyses, based on state machine replication via the Paxos consensus protocol at the level of the POSIX socket API [24]. While RepFrame does not require rewrite rules, it puts important restrictions both on the types of applications that it can run (i.e. it targets server programs only), and on the number of native and lightweight analyses that have to be run (over half, in order to achieve distributed consensus on the socket API). *FreeDA* does not impose any of these restrictions.

Bunshin [46] also uses a multi-version execution approach to deploy incompatible dynamic analyses, focusing on reducing overhead by distributing analysis checks across multiple versions. Compared to *FreeDA*, Bunshin exhibits the following limitations: (1) it only supports a limited class of analyses, those which instrument the program with independent runtime checks, so tools such as Valgrind are out-of-scope; (2) it requires internal knowledge of the underlying tools, e.g., to know that one could distribute the analysis checks across functions for ASan and MSan, and across sub-sanitizers for UBSan [43]; and (3) it reduces the overhead of the analysis substantially (e.g., from 107% to 47.1% for ASan and from 228% to 94.5% for UBSan), but this is still too high to be acceptable in most cases in production. We note that the high overhead is partially due to Bunshin's focus on preventing attacks at runtime, which is not what *FreeDA* aims to achieve.

To make an analysis deployable in production, one needs to make it fast. Some analyses already meet this criterion. Fast, limited-in-scope analyses such as stack canaries [22] are now enabled by default in most compilers, while more general analyses such as address sanitization [38] can already be used in production in some situations, e.g., for certain interactive applications. Prior work has looked at designing dynamic analyses with a low overhead through a variety of means, such as hardware support [34, 48], parallelization [33, 45] and trip wires [30]. In contrast, our approach allows one to use *any* available analyses, and apply them concurrently on the same program execution.

The idea of running dynamic analyses in parallel with a native version of the program is an old one. In particular, Patil and Fisher [35] were the first ones to propose the use of what they call a *shadow process* that runs in parallel with the native application and performs additional dynamic analysis checks. Speck [33] extends this idea by running the native application ahead speculatively and forking multiple analysis instances, and SuperPin [45] by simultaneously executing distinct timeslices of the native program under instrumentation. Aftersight [21] employs a record-replay strategy similar to *FreeDA*, but operating at the virtual machine (VM) level. A VM-based architecture has the advantage of supporting the analysis of kernel code, but it makes it harder to deploy user-level software, such as interactive applications. *FreeDA* draws inspiration from all this body of prior work and shares much of their high-level ideas, but it distinguishes itself by enabling the deployment of *incompatible stock dynamic analysis tools*. This separation of concerns between the experts writing the dynamic analyses and the runtime platform that allows their concurrent deployment with low overhead is an important contribution of *FreeDA*. Although the analyses implemented by prior work are often as powerful as those of popular tools like Valgrind, in practice they miss usability features and optimizations that prevent their adoption in practice. In contrast, *FreeDA* does not provide its own implementation of dynamic analyses, and instead allows the best existing analyses to be deployed transparently.

FreeDA is a multi-version execution system (MVE) [17, 20, 23, 27, 31, 37, 44, 47] which enables the low-overhead deployment of incompatible stock dynamic analysis tools. Unlike prior multi-version execution systems, in which all versions run at roughly the same speed, *FreeDA* has to support versions that run significantly slower than the native one. Variant-based competitive parallel execution

creates variants of a sequential algorithm with different performance characteristics (e.g., using different heuristics), run them in parallel, and use the results of the algorithm that returns first [42]. As for *FreeDA*, the different variants have different speeds. But unlike *FreeDA*, the approach is not concerned with the slow variants falling behind, as the different variants run unsynchronized, and as soon as one variant returns a result, the others are terminated.

Encoding system-call changes in MVE, including those introduced by dynamic analyses, was addressed in previous work using Haskell [31], BPF filters [28], or domain specific languages [36]. This paper introduces a detailed taxonomy of the system-call changes introduced by DA tools, but its main focus is on deploying these tools efficiently in production.

8 CONCLUSION

While the last decades have seen the design of a diversity of effective dynamic analysis tools, many of them are incompatible with one another, and their overhead is often too large for them to be applied in production. *FreeDA* is a novel system that can be used to transparently deploy *incompatible stock dynamic analysis tools*, while providing close to native performance to users on realistic scenarios. In particular, we have shown that *FreeDA* can be used to run popular analyses such as those implemented by Valgrind and Clang's compiler sanitizers on both high-performance server applications, such as Memcached and Redis, and interactive applications, such as Git and OpenSSH. We believe that *FreeDA* can significantly increase the impact of dynamic analysis techniques, and thus make software more reliable.

ACKNOWLEDGEMENTS

We thank Paul-Antoine Arras, Andrea Mattavelli and the anonymous reviewers for their useful feedback. This research was generously sponsored by the EPSRC through the Early-Career Fellowship EP/L002795/1.

REFERENCES

[1] CMake, the cross-platform, open-source build system. https://github.com/Kitware/CMake. Accessed: 2017-04-03.
[2] git –local-branching-on-the-cheap. https://git-scm.com/. Accessed: 2017-04-03.
[3] Git Source Code Mirror. https://github.com/git/git. Accessed: 2017-04-03.
[4] HAProxy — The Reliable, High Performance TCP/HTTP Load Balancer. http://www.haproxy.org/. Accessed: 2017-04-03.
[5] htop - an interactive process viewer for Unix. http://hisham.hm/htop/. Accessed: 2017-04-03.
[6] memcached — a distributed memory object caching system. http://memcached.org/. Accessed: 2017-04-03.
[7] memcached development tree. https://github.com/memcached/memcached. Accessed: 2017-04-03.
[8] memtier — NoSQL Redis and Memcache traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark. Accessed: 2017-04-03.
[9] NGINX | High Performance Load Balancer, Web Server, & Reverse Proxy. https://www.nginx.com/. Accessed: 2017-04-03.
[10] OpenSSH homepage. http://www.openssh.com/. Accessed: 2017-04-03.
[11] OpenSSL Cryptography and SSL/TLS Toolkit. https://www.openssl.org/. Accessed: 2017-04-03.
[12] OpenSSL FAQ — Why does Valgrind complain about the use of uninitialized data? https://www.openssl.org/docs/faq.html#PROG14. Accessed: 2017-04-03.
[13] Redis homepage. http://redis.io/. Accessed: 2017-04-03.
[14] The Expect Home Page. http://expect.sourceforge.net/. Accessed: 2017-04-03.
[15] twemproxy — A fast, light-weight proxy for memcached and redis. https://github.com/twitter/twemproxy. Accessed: 2017-04-03.
[16] wrk2 — A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2. Accessed: 2017-04-03.
[17] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: probabilistic memory safety for unsafe languages. In *PLDI'06*.
[18] Derek Bruening and Qin Zhao. 2011. Practical Memory Checking with Dr. Memory. In *CGO'11*.
[19] Cristian Cadar and Petr Hosek. 2012. Multi-Version Software Updates. In *HotSWUp'12*.
[20] Liming Chen and Algirdas Avizienis. 1978. N-version programming: A Fault-tolerance approach to reliability of software operation. In *FTCS'78*.
[21] Jim Chow, Tal Garfinkel, and Peter M. Chen. 2008. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *USENIX ATC'08*.
[22] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *USENIX Security'98*.
[23] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. 2006. N-variant systems: A secretless framework for security through diversity. In *USENIX Security'06*.
[24] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. 2015. Paxos Made Transparent. In *SOSP'15*.
[25] Heming Cui, Rui Gu, Cheng Liu, and Junfeng Yang. 2015. Repframe: An efficient and transparent framework for dynamic program analysis. In *ApSys'15*.
[26] Reed Hastings and Bob Joyce. 1992. Purify: Fast Detection of Memory Leaks and Access Errors. In *USENIX Winter'92*.
[27] Petr Hosek and Cristian Cadar. 2013. Safe Software Updates via Multi-version Execution. In *ICSE'13*.
[28] Petr Hosek and Cristian Cadar. 2015. Varan the Unbelievable: An efficient N-version execution framework. In *ASPLOS'15*.
[29] K. Koning, H. Bos, and C. Giuffrida. 2016. Secure and Efficient Multi-Variant Execution Using Hardware-Assisted Process Virtualization. In *DSN'16*.
[30] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2016. DoubleTake: Fast and Precise Error Detection via Evidence-based Dynamic Analysis. In *ICSE'16*.
[31] Matthew Maurer and David Brumley. 2012. TACHYON: Tandem Execution for Efficient Live Patch Testing. In *USENIX Security'12*.
[32] Nicholas Nethercote and Julian Seward. 2003. Valgrind: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science* 89, 2 (2003).
[33] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. 2008. Parallelizing Security Checks on Commodity Hardware. In *ASPLOS'08*.
[34] Jeffrey Oplinger and Monica S. Lam. 2002. Enhancing Software Reliability with Speculative Threads. In *ASPLOS'02*.
[35] Harish Patil and Charles Fischer. 1995. Efficient Run-time Monitoring Using Shadow Processing. In *AADEBUG'95*.
[36] Luís Pina, Daniel Grumberg, Anastasios Andronidis, and Cristian Cadar. 2017. A DSL Approach to Reconcile Equivalent Divergent Program Executions. In *USENIX ATC'17*.
[37] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. 2009. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *EuroSys'09*.
[38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC'12*.
[39] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer—data race detection in practice. In *Workshop on Binary Instrumentation and Applications*.
[40] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX ATC'05*.
[41] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *CGO'15*.
[42] Oliver Trachsel and Thomas R. Gross. 2010. Variant-based competitive parallel execution of sequential programs. In *CF'10*.
[43] UBSan 2017. Undefined Behavior Sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. (2017).
[44] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. 2016. Secure and Efficient Application Monitoring and Replication. In *USENIX ATC'16*.
[45] Steven Wallace and Kim Hazelwood. 2007. SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance. In *CGO'07*.
[46] Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. 2017. Bunshin: Compositing Security Mechanisms through Diversification. In *USENIX ATC'17*.
[47] Hui Xue, Nathan Dautenhahn, and Samuel T. King. 2012. Using Replicated Execution for a More Secure and Reliable Web Browser. In *NDSS'12*.
[48] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. 2004. iWatcher: Efficient Architectural Support for Software Debugging. In *ISCA'04*.