Imperial College London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Accelerating Dynamic Analyses Tools Through Multi-Version Execution

Author:
Anastasios Andronidis

Supervisor:
Cristian Cadar

Submitted in partial fulfillment of the requirements for the MSc degree in MRes of Imperial College London

September 2016

Abstract

Dynamic analysis tools such as Valgrind and compiler sanitizers are effective at finding challenging bugs and security vulnerabilities. However, they incur a high overhead, which typically prevents them from being used in production.

This report addresses the ambitious goal of running such dynamic analysis tools in production, without requiring any modifications to the dynamic analysis tools nor the deployed system, and without adding significant runtime overhead. This is accomplished using multi-version execution, in which the dynamic analyses are run alongside the production system.

In particular, we show how existing unmodified dynamic analyses implemented by Valgrind and Clang's sanitizers can be used to check high-performance servers such as Nginx and Redis.

Declaration

This thesis presents my own original research and all the sources have been quoted and referenced accordingly.

Parts of this work are currently under submission in a conference.

Acknowledgements

First and foremost, I would like to thank my supervisor Dr Cristian Cadar for all his support, advice and help throughout this year. His guidance showed me the way of conducting proper research and helped me through the difficulties in the process, which allowed me to learn a lot.

Secondly, I would like to thank Dr Luís Pina for his help and mentoring on the technical challenges with *Varan*, and Mr. Daniel Grumberg for his excellent work on the *Divela* compiler.

Finally, I would like to thank my family for their endless support, and last but not least my friend Pantazis Deligiannis for his pivotal support, advice and help.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

As digital services and electronics have become an essential, if not mandatory, part of our modern civilization, the existence of correctly behaving and reliable software has become crucial for their normal operation. We delegate more and more responsibilities to our software and this has led to increasingly more complex systems [Cav13; Lag+15]. This complexity leads to more defects and experience from practice has shown that creating flawless software is extremely hard and sometimes even impossible. The consequences of faults in software systems can range from mildly annoying to catastrophic [Tas02], involving serious injuries or even lives lost [LT93; Neu+80]. Due to these facts, a whole new branch in computer science has emerged which is dedicated to study different options and possibilities that could assist software developers in increasing their software quality.

In the recent years, research managed to produce tools that significantly assist software developers to expose rare and hard-to-find bugs in their code. When such tools are used on applications, end results do not change but extensive internal health checks during runtime are added. Unfortunately, this is a significant trade-off between performance and security as those checks are not coming for free but are extremely useful for finding defects. Such tools are known as Dynamic Analyses tools.

In this report we discuss how can we improve this performance penalty of Dynamic Analyses tools and allow software developers to deploy them even in productions systems where performance is of the utmost importance. To achieve this we first need to review the ideas of N-Version Programming (NVP), which we discuss briefly in section 1.2, and then introduce ourselves to N-Version Execution (NVX) a component of NVP.

At a high level, N-Version Execution is the system that allows multiple software instances to synchronize their execution and their side-effects to the external environment. For instance, under NVX we are allowed to execute simultaneously two programs that send one email each, where the end user will receive only one. In Chapter 2 we dive deeper into the design and implementation of N-Version Execution. We also focus our discussion into some key concepts and features of the tool

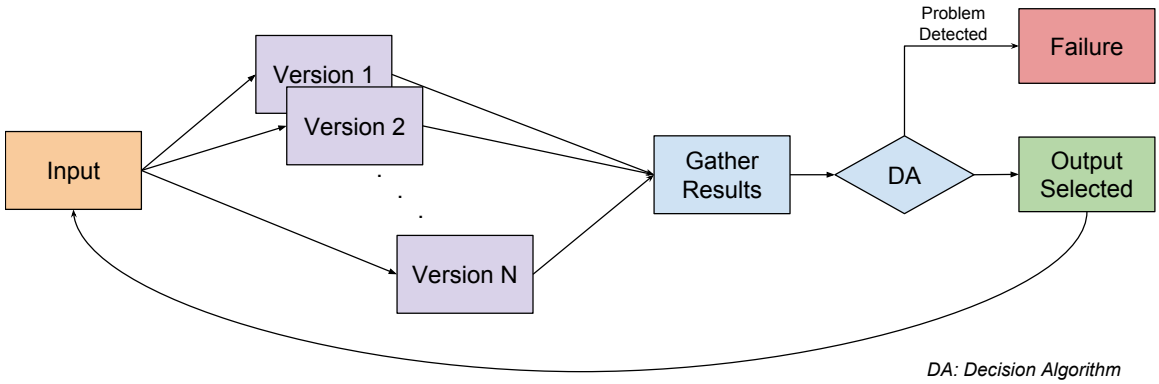

Figure 1.1: Implementation of an N-Version Programming system.

Varan [HC15] which we later use in all of our experiments.

In Chapter 3 we discuss how N-Version Execution can be combined with Dynamic Analyses tools and allow the deployment of both heavyweight and lightweight Dynamic Analyses in production systems. We see examples of various such tools like Valgrind and LLVM Sanitizers, executed together with popular database: Redis and the web server: Nginx. We also extend our approach on how we can handle cases were the executed applications are the same application but configured differently. For example, a Redis instance that keeps data only in-memory synchronizes execution with a second Redis instance that writes its data in a persistent storage medium.

Lastly, in Chapter 4 we discuss how NVX can tolerate small differences in execution between the executed applications. In the previous paragraph we mentioned cases were we execute Redis with Valgrind. Synchronizing such two applications (Redis and Redis under Valgrind) is far from trivial as the behaviour of Redis under Valgrind is similar to Redis but not identical. A proper NVX system should be flexible enough to tolerate such differences and we show how we can develop the appropriate mechanisms to tolerate such differences.

1.2 A note on N-Version Programming

The primary objective of N-Version Programming (NVP) is to mask the effects of software defects during execution. To achieve that, NVP requires the independent creation (created from different teams of programmers) of functionally equivalent software from the same specification.

When such independent software instances are obtained, NVP describes how the execution of those instances can be synchronized and how their output can be combined to tolerate defects. Figure 1.1 shows the overview of NVP where multiple instances of the same software operate on the same input data and produce results that NVP compares. If any instance produces an incorrect result, then the system can either mask the problem by using the output from other correctly behaving instances or exit with an error. To be noted that if the majority of instances produce the same wrong results then we allow a defect to progress undetected.

The core ideas of NVP are captured in the following principle:

> "[The] independence of programming efforts will greatly reduce the probability of identical software faults occurring in two or more versions of the program" – [CA78]

Throughout the years many researchers contributed to the ideas of NVP, especially by creating tools and frameworks that made NVP much more approachable to developers. However, one major dispute on the aforementioned funding principle of NVP changed the field quite drastically.

Knight and Leveson conducted a large scale experiment in 1986 [KL86; BKL90] to evaluate the basic NVP principle of fault independence. They found that programmers tend to make the same mistakes independently. Those results initiated a series of disputes [KL90] with strong debates between the researchers.

Recent research has been much more positive. Due to advancements in compilers, and other automatic generating program techniques, researchers has shown that we can create programs with specific properties that under NVP can complement each other. For instance, Nagy et al. [NFA06] show that by generating, though compiler options, programs with different memory layouts (stack growing at a different direction) would be extremely hard if not impossible to exploit a memory vulnerability common in both versions. Also in our experiments, when using Dynamic Analysis tools we are guaranteed that specific types of bugs (e.g. out of bound memory reads) will be captured while the native application will continue execution without noticing any issues. Thus if a malicious activity spotted, Dynamic Analysis tools would inform us of the problem.

In general, the Knight and Leveson study was important for the future development of the field as they motivated researchers to look further and more carefully on how software diversity can be properly generated.

Chapter 2

N-Version Execution

N-Version Execution (NVX) is the software and/or hardware component that manages the N individual software versions of NVP. More specifically, NVX is the execution environment and the system that handles the input, output, scheduling of processes or threads, and compares the behavior of the N software instances. Note that NVX does not impose any specific logic now how mismatching outputs should be handled.

Even though N-Version Execution (NVX) is only a part of N-Version Programming, there is a significant interest in the applicability of such frameworks into many research areas such as dynamic software updates [Qia+15], software recovery [HC13a; CH12], performance improving [TG10] and many more. In this chapter we discuss the basic attributes of NVX frameworks through the Varan system [HC15].

2.1 Varan

Even though the details of the tool are presented in great detail in [HC15], we review some of its most important concepts as *Varan* is the core tool used in all of our experiments later on.

Varan is an NVX framework that combines selective binary rewriting with an event-streaming architecture. It significantly reduces performance overhead comparing to other state of the art systems that rely on alternative methods like *ptrace* or intrusive kernel modifications.

In order to understand the internals of *Varan* we would like to guide the discussion through an example. In Figure 2.1 an example application invokes the `write()` function in order to write some data in a file. Often, such functions are defined in external libraries that the system loads as shared objects in the application's memory space. Then the library invokes some lower level functions, known as system calls, that communicate information directly to the kernel of the operating system. As a first approximation the kernel receives the call, executes it, returns the results back to the application and the application execution continues normally.

Varan uses binary rewriting techniques to intercept all the calls targeting the kernel (i.e. system calls). When an application is executed under *Varan*, an extra library is loaded in memory and then *Varan* parses the application memory to find

all system call invocations. Every time *Varan* finds a system call invocation, it deletes it and redirect the execution inside the *Varan* related library. Now when the application executes a `write()` function, the execution path is intercepted as shown in Figure 2.2.

This *Varan* related library contains functionality to synchronize the system calls between different software instances executed concurrently under *Varan*. In every system call made by the application, *Varan* stores the arguments and the return values in a shared ring buffer as shown in Figure 2.2.

Varan allows only one application publish to the shared ring buffer, but allows multiple applications to consume. We refer to the publisher application as the *leader* and the consumer as the *follower* (or followers in case of multiple). Figure 2.3 shows how a leader executes a `write()` which is intercepted by *Varan* and stored in the shared ring buffer. Then when a follower executes the same code and reaches the same function call, *Varan* detects that this system call is already made by the leader and returns the stored values from memory instead of calling the kernel. When all followers consumed a specific system call result, *Varan* deletes the entry to free space for the leader to continue publishing.

The overall architecture of *Varan* is shown in Figure 2.4. The user only interacts with the leader and the followers synchronize behind the scenes. Due to the shared ring buffer all followers emulate their external behaviour and thus *Varan* gives the illusion to the user that only the leader is executing. On the other hand all followers maintain the same internal state and have the impression that they are the leader. Currently no communication between the instances is allowed.

2.1.1 Reduced system time in followers

It should be apparent from Figure 2.2 that *Varan* adds some extra overhead. The leader application has to store its system call information into a shared ring buffer before and after calling the kernel. This extra overhead is due to two important factors:

1. Copying the arguments and return values to the shared ring buffer (e.g. think of a `read()` system call that returns a large string)

2. Atomic and synchronization operations on the ring buffer to avoid races between the applications running under *Varan*.

To be noted that due to the fact that all these actions are in memory, the imposed overhead is quite small [HC15].

It might come as a surprise but *Varan* is also able to speed up some of its executed applications. This speed up effect is only observable at the followers and not at the leader. As we described in Figure 2.3 the follower usually does not execute any real system calls, rather it consumes the stored results from the shared ring buffer. Due to this mechanism, the follower does not invoke any kernel interactions and thus has minimal *system time*.

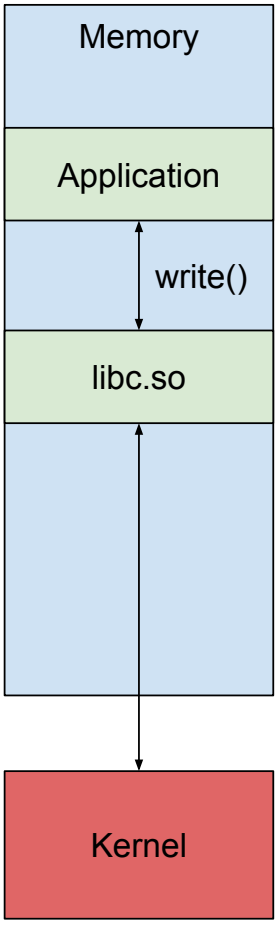

Figure 2.1: An application invoking a write() system call.

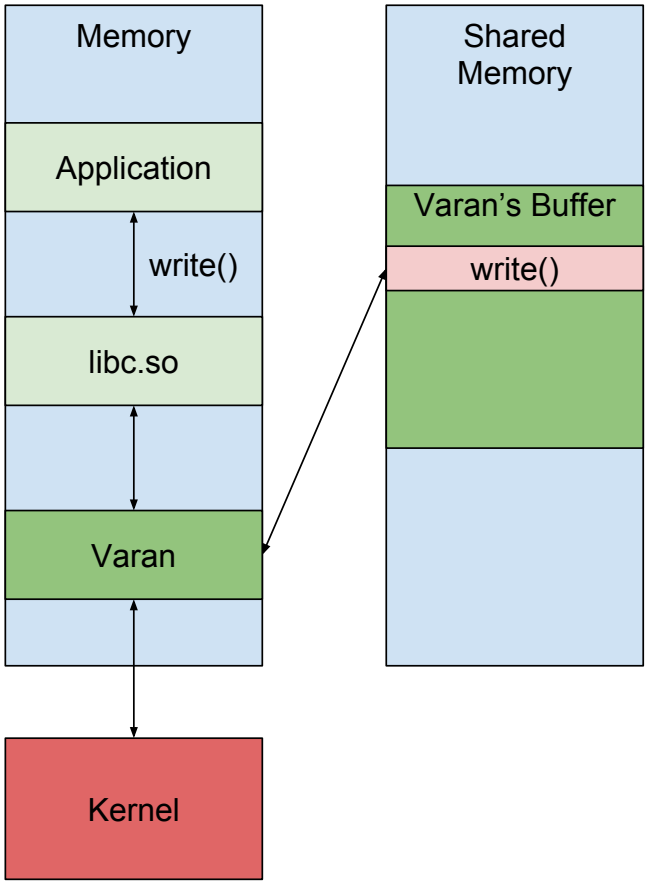

Figure 2.2: The leader invoking a write() system call.

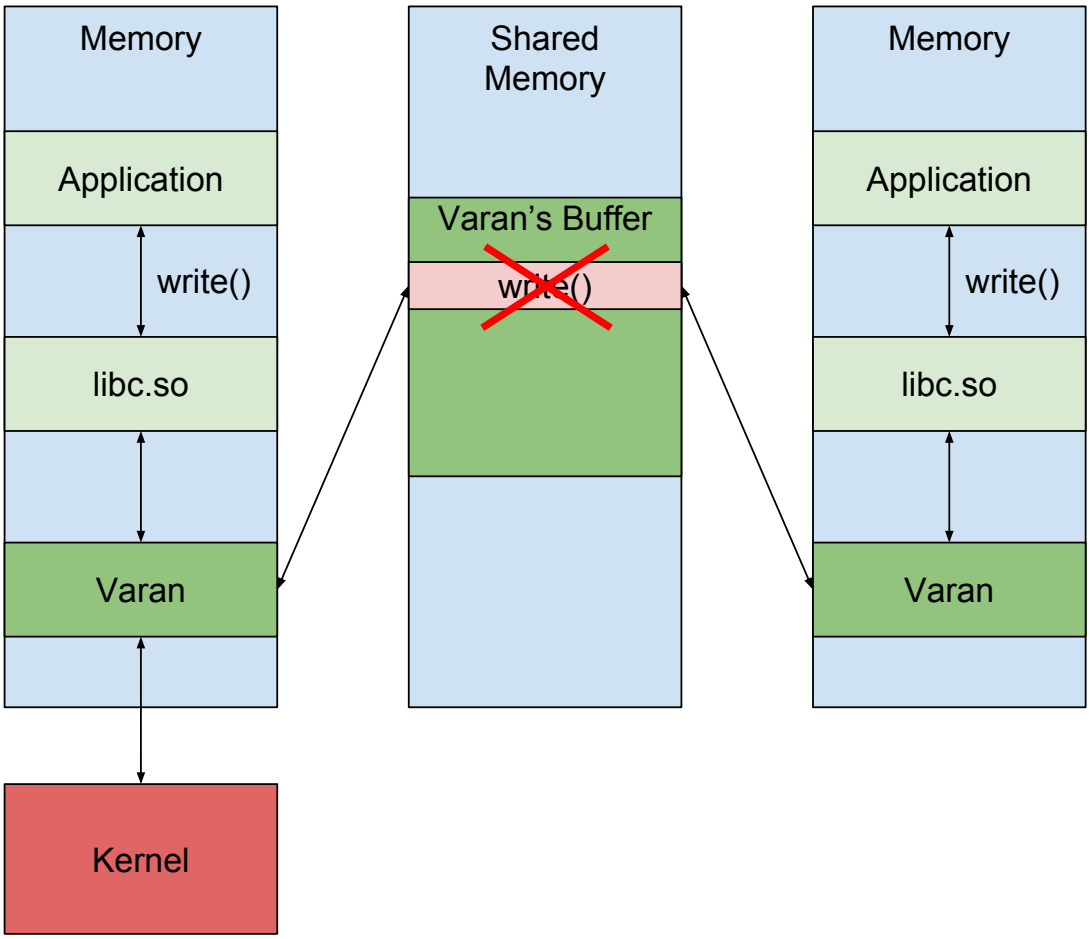

Figure 2.3: The follower consumes the results of a write() system call.

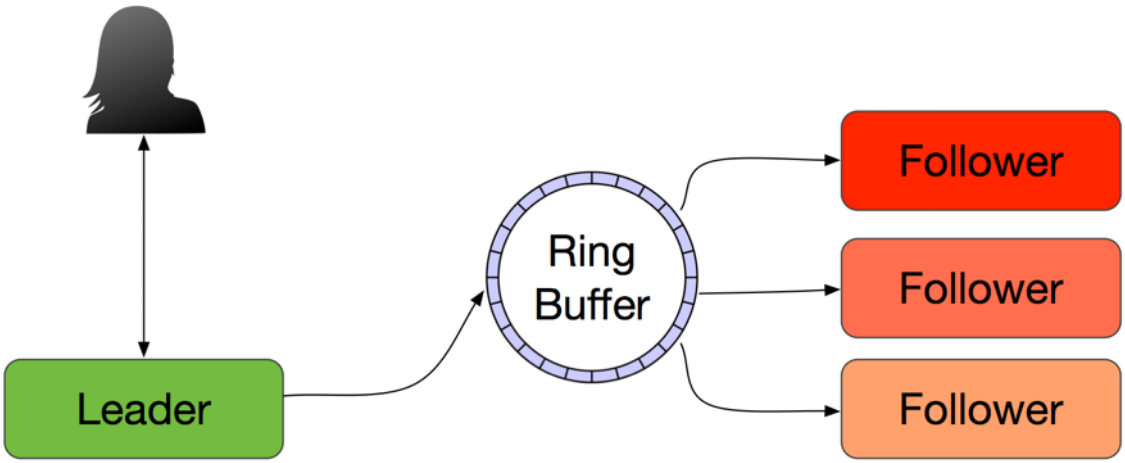

Figure 2.4: Architecture of Varan. The user only interacts with the Leader where the Followers synchronize silently behind the scenes.

In a short experiment we saw that *system time* can be reduced significantly. We tested two Nginx instances (as a leader and follower) under *Varan* for 50 seconds of emulated traffic (using the *wrk2* tool). The system time reported from `/proc/<pid>/stat` for the leader was 50% (i.e. 25 seconds) and for the follower was 10% (i.e. 5 seconds).

2.1.2 Ignoring-File-Paths Extension

During the project various bug fixes and enchantments were made to accommodate new requirements and use cases. In this section we shall discuss a specific feature that its purpose is to allow followers to operate on predefined files or directories independently from the leader. This feature is important for understanding the experiments conducted in the following chapters. We will also refer to this feature again when we shall introduce software divergences more formally.

Let us assume two programs in which the first one does some kind of computation from a given input, and the second operates identically with the only difference that it writes some internal debug messages in a log file. With this newly added feature, we can inform *Varan* that any system call coming from the follower and targets the aforementioned log file, are safe to be executed even no similar system calls exist in the ring buffer.

From the implementation point of view, the user informs *Varan* (through some special flag option) which are the file names or directories *Varan* should ignore. Then *Varan* monitors any system call that operates on files. System calls that return file descriptors and accept a string value as one of their arguments, are candidates for further inspection (e.g. `open()`). If the string value argument matches any user given file or directory then *Varan* executes the system call and stores the returned file descriptor in memory. No operation happens at the ring buffer. From that point onwards if any other system call has as its argument the stored file descriptor (e.g. `write()`), will to be executed normally without *Varan* looking at the ring buffer first.

In code 2.1 we can see an example of how *Varan* can ignore files (we omit file separation, checks and library inclusions for the sake of simplicity). As we can see the leader does not perform any interaction with the file system. On the other hand, the follower opens a file, writes some information and then closes it. Without the discussed feature, *Varan* would have stopped the execution as the leader's sequence of system calls does not match the follower's.

There is a subtle problem though which is demonstrated in Code 2.2. *Varan* will be able to match the first `open()` system call in both leader and follower. Then the second `open()` in the follower will be ignored and the kernel will return the valid file descriptor 5 to the follower. But when we reach the third `open()` in line 13, *Varan* will return a file descriptor of number 5 from the ring buffer, while the follower would expect the number to be 6. As the follower issues an extra `open()`, the correct value of file descriptor returned from the kernel should be by 1 larger.

To solve this issue we use `dub()` to copy all ignored file descriptors to values above 512. In this way we keep low values free for the kernel to use for system calls

Listing 2.1: Ignore file writes in the follower

```
// Leader's code
int main(int argc, char **argv) {
  return 0;
}

// Follower's code
int main(int argc, char **argv) {
  FILE *f = fopen("a.txt", "w");
  const char *text = "hi";
  fprintf(f, "%s\n", text);
  fclose(f);

  return 0;
}

// To execute Varan
// vx --ignore-files a.txt leader.exe follower.exe
```

that we want to keep synchronized. This is not a perfect solution as scalability issues can arise fast (e.g. if the user opens many files). Ideally we would need to keep a table to match the file descriptor values from the leader to the follower. We did not observe any problems until now in all of our experiments.

We do also support the same functionality from the leaders perspective. When the user wants to ignore a file that only the leader operates upon, then *Varan* uses `dub()` again in the same way described above, executes the system call and does not store the system call at the shared ring buffer.

Ignoring files from the leader and the follower gives us great flexibility as it allows us to explicitly request followers with additional functionality from the leader so we can use it in our advantage. More specifically, we will see in the next chapter how we can leverage this feature to execute two Redis processes from which, one is keeping data only in memory (and thus is fast) and the second writes its data in a file for persistent storing.

Listing 2.2: Problem with naively ignoring files

```
// Leader's code
int main(int argc, char **argv) {
  FILE *f1 = fopen("1.txt", "w"); // f1 == 4
  FILE *f2 = fopen("2.txt", "w"); // f2 == 5
  ...
  return 0;
}

// Follower's code
int main(int argc, char **argv) {
  FILE *f1 = fopen("1.txt", "w"); // f1 == 4
  FILE *ig = fopen("ignore_me.txt", "w"); // ig == 5
  FILE *f2 = fopen("2.txt", "w"); // f2 == 6!!!
  ...
  return 0;
}

// Varan will be executed as
// vx --ignore-files ignore_me.txt leader.exe follower.exe
```

Chapter 3

Accelerating Dynamic Analyses Tools

Dynamic analyses tools such as compiler sanitizers [Ser+12; SS15; SI09] and Valgrind [NS03] are readily applied directly to program binaries and are effective at finding challenging bugs and security vulnerabilities. These tools have gathered an impressive number of trophies, in the form of serious bugs and security vulnerabilities discovered in popular software systems. Nevertheless, due to the high runtime overhead that they introduce, dynamic analysis tools are typically not used in production systems and stay restricted to offline testing runs, which exercise a limited, often artificial, set of program executions.

In this chapter we shall discuss the ambitious goal of running such dynamic analysis tools in production, without requiring any modifications to the dynamic analysis tools nor the deployed system, and without adding significant runtime overhead. This goal is achievable by running the dynamic analyses tools alongside the production system under *Varan* [HC15].

To achieve this goal we need to address two major challenges:

1. Tolerate expected divergences between leader and followers
2. Avoid filling the shared ring buffer

When an application is running under a dynamic analyses tool (e.g. Valgrind) will behave slightly differently (i.e. will have different system call patterns). Thus we need a mechanism in *Varan* that will be able to understand when those different patterns (i.e. divergences) between leader and follower are due to the dynamic analyses tool and not due to a failure or security vulnerability. The mechanism addressing this challenge will be discussed extensively in chapter 4. For now we will assume that there exists such a mechanism and is able to understand and handle the differences in system calls. In this chapter we focus solely in challenge number 2.

As dynamic analyses tool have to perform various checks during the execution of an application, users experience a significant slowdown. This affects *Varan* too as if the followers are much slower than the leader then the shared ring buffer ultimately will become full, prohibiting the leader to run in full speed. We will discuss the details of this issue and give a possible solution in section 3.1.

We shall refer to the system that combines *Varan* with the solutions for chal-

lenges 1 and 2, as *FreeDA*.[1] We show that *FreeDA* is applicable in several common scenarios where applications can be used behind a load balancer. In particular, we show how existing unmodified dynamic analyses implemented by Valgrind and Clang's sanitizers can be used to check high-performance servers such as Nginx and Redis.

Disclaimer: In the work presented in our under submission paper, we also applied *FreeDA* in interactive applications like Git, OpenSSH, and HTop where the user's input is slow enough to not saturate the shared ring buffer. We do not include details of those additional experiments as this part of the work is not mainly driven by the author of this report.

Lastly, *FreeDA* can deploy multiple analyses concurrently due to the inherited property of *Varan* to execute multiple followers simultaneously. Interestingly, such analyses might be incompatible with one another (e.g. ASan with MSan).

Of course, *FreeDA* pays a price in terms of utilization overhead. However, as it has been argued in prior work on multi-version execution, many cores are currently left idle, and these idle cores also consume significant energy [BH07]. Furthermore, the decentralized architecture of *Varan* allows the system to simply terminate the execution of the dynamically-checked followers when resources are scarce.

3.1 Reducing Latency and Scaling Throughput

The two most critical performance factors for server systems are latency (i.e. how fast an operation can be completed) and throughput (i.e. how many operations can be completed per unit of time).

Dynamic analysis tools increase latency and decrease throughput significantly. When possible, throughput decrements can be successfully mitigated by various well established scaling techniques (e.g. load balancing), but increases in latency are just unacceptable as latency is strongly linked with the algorithm or the design of the system.

Fortunately, due to *Varan*'s architecture we can expose a fast leader to the user and give the impression of low latency while the slow followers synchronize in the background. There is a catch though; *Varan* will maintain low latency as long as there is enough free space in the shared ring buffer. If the shared ring buffer gets full, then the leader will block until a follower consumes some stored system call data.

To make things concrete, let us think of the following example. Given an application that can sustain a maximum of 10 requests per second with a 100ms latency per request, we can produce a second application operating under ASan which will have 200ms latency and thus 5 requests per second as maximum throughput (see Figure 3.1). Putting the first application as the leader and the second one as follower, we can export the low latency of the leader up to 5 requests per second as

[1] The name *FreeDA* is a word play on *Free Dynamic Analysis*.

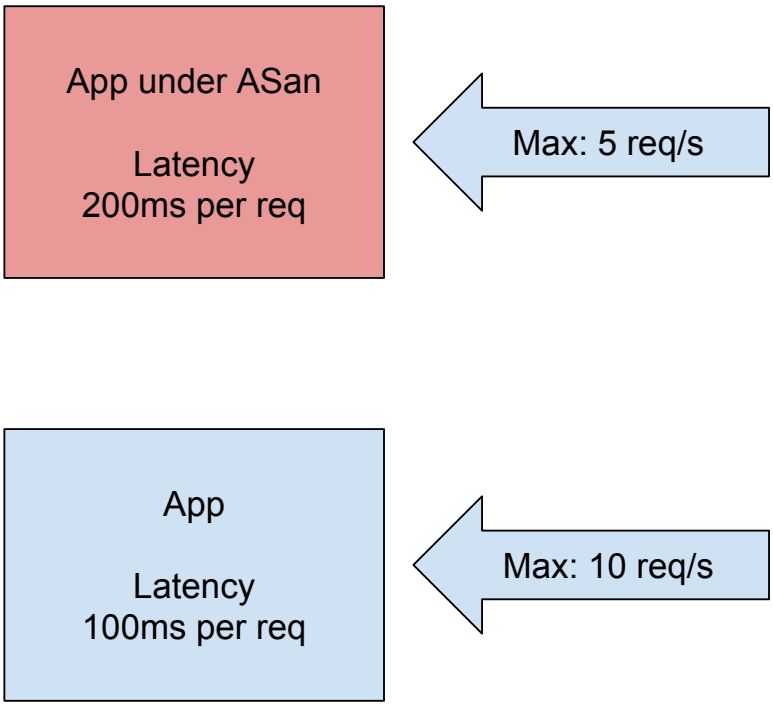

Figure 3.1: The maximum throughput and latency of an example application under ASan.

shown in Figure 3.2. If the throughput increases above the maximum throughput of the slowest follower then the shared ring buffer becomes full and we export the latency of the slowest follower as shown in Figure 3.3.

Another way to think about how the shared ring buffer can become full is by imaging a water tank with an tube providing water and a sink allowing the water to escape. If the sink supports up to 5 liters of water per second to escape the tank then as long as the tube does not provide more than this threshold we are guaranteed that the tank will never get full. On the other hand if the tube provide even the tiniest amount more, then the tank will eventually get full independently how large the tank is.

To sum up:

1. *Varan* can not increase throughput (except under the conditions discussed in 2.1.1), on the contrary maximum throughput is bounded from the slowest follower.
2. Independently of its size, the shared ring buffer will eventually become full if the incoming throughput exceeds the maximum throughput of the slowest follower.
3. As long as the shared ring buffer is not full, we expose the latency of the leader.

This is why *Varan* naturally complements throughput scaling techniques like load balancers. *FreeDA* combines *Varan* with state of the art load balancing systems to allow the deployment of dynamic analysis tools in production with very little cost.

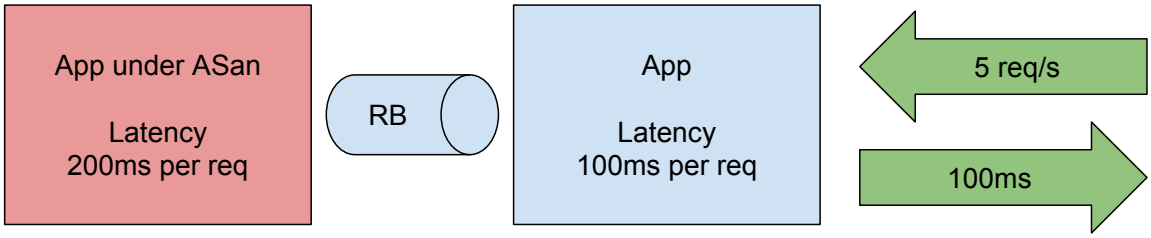

Figure 3.2: Maximum throughput that can be maintained to export native latency.

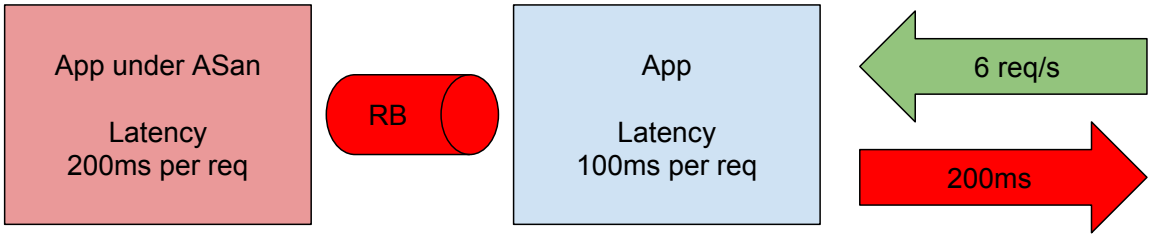

Figure 3.3: If we exceed the maximum throughput of the slowest follower then we suffer from a latency degradation.

3.2 Experimental Setup

We applied *FreeDA* to two high-performance widely-used network servers, Nginx and Redis, used in conjunction with load balancers HAProxy[2] and Twemproxy.[3] To benchmark Nginx we used *wrk2*[4] and for Redis we used *memtier*.[5]

Nginx[6] is a highly-popular reverse proxy server, often used as an HTTP web server, load balancer, or cache. We used version 1.11.2 in our experiments. We benchmarked Nginx with *wrk2* version 4.0.0, which is a highly-configurable HTTP performance benchmark that can measure latency while keeping a given throughput.

Redis[7] is a high-performance in-memory key-value data store, used by many well-known services. We used version 3.0.7. We benchmarked Redis with *memtier* version 1.2.7 which is a highly-configurable key-value store benchmark that can measure latency while keeping a given throughput.

We conducted our experiments on a cluster of three machines, all located on the same rack and connected by a 1Gb Ethernet link. For convenience, we name the machines M1, M2 and M3:

M1: a machine with two 2.50GHz Intel Xeon E5-2450 v2 CPUs (each CPU with 8 physical cores, 16 logical) with 188G of RAM, and running 64-bit Ubuntu 14.04 (kernel version 3.16.0-34).

[2] `http://www.haproxy.org/`
[3] `https://github.com/twitter/twemproxy`
[4] `https://github.com/giltene/wrk2`
[5] `https://github.com/RedisLabs/memtier_benchmark`
[6] `https://www.nginx.com/`
[7] `http://redis.io/`

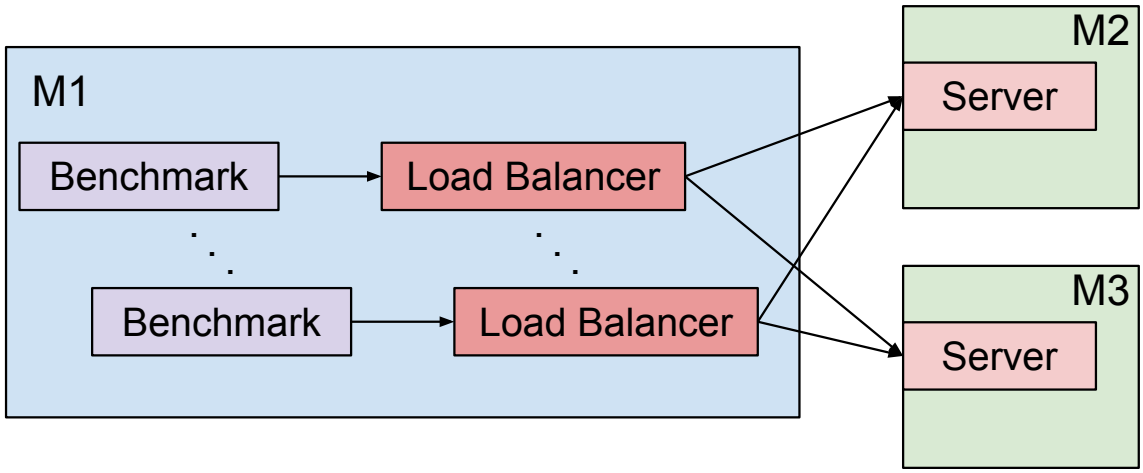

Figure 3.4: Overall architecture of the experiments.

M2, M3: a pair of machines 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).

All the results presented are the average of 5 consecutive runs that generate traffic for 50 seconds. Additionally we used `taskset` to pin processes to specific CPUs to avoid noise from the scheduler. For every benchmark iteration all services were restarted to include cold start conditions.

The overall architecture of how the services were deployed, is shown in Figure 3.4. For the direct connection experiments presented in section 3.3 the *Load Balancer* component is omitted.

3.3 Baseline Experiments

We used *FreeDA* to deploy an ASan version of Nginx and benchmarked a direct connection to the server (i.e. no load balancing). We ran *wrk2* on M1 to connect to an Nginx server running on M2. We configured *wrk2* to transfer a 1KiB file for 50 seconds using 4 threads and 80 open connections. We ran three versions of the server: native, ASan, and *FreeDA* with a native version as a leader and an ASan version as a follower.

We first measured the maximum throughput that saturates ASan. To do so, we configured *wrk2* to achieve increasing throughput values, from 20K req/s, with a step of 4K req/s, until ASan could not maintain the throughput. We measured the latency reported for each throughput value.

Figure 3.5 shows the results. ASan can maintain a throughput of only 32K req/s (the native version achieves a maximum of 48K req/s). *FreeDA* achieves a higher throughput, up to 36K req/s, because, as we explained in section 2.1.1, the sanitized follower saves the time spent inside the kernel when executing each system call by reading the results directly from the shared ring buffer. However, at this throughput,

the buffer starts to become full and the latency of *FreeDA* increases to that of the ASan follower.

As expected, ASan increases the latency as the throughput increases, resulting in a latency increase of 35.2% at 32K reqs/s. *FreeDA* increases the latency by just 7.4% at 32K reqs/s (and even less for lower throughputs).

Downloading a file is an I/O-bound benchmark. The server performs little work, spending most time waiting for I/O. As a result, ASan has few checks to perform and thus introduces little overhead. To stress ASan, we ran a second experiment, in which we configured Nginx to use protocol-level compression (*Gzip*) to serve a file of 2KiB containing random data. As before, we ran the experiment for increasing throughputs, starting at 7K req/s with a 1K req/s step, until ASan could not maintain the throughput.

Figure 3.6 shows the results. Compression decreases the throughput significantly, and ASan can maintain only 10K req/s (the native version achieves a maximum of 14K req/s). Note that the latency is similar between figures 3.5 and 3.6, but we are reporting different throughput levels. In other words, in figure 3.5 latency is dominated by *system time* due to I/O, where in figure 3.6 latency is mostly due to compression and ASan.

ASan increases the latency significantly, adding an overhead of 61.2% at 10K req/s. *FreeDA* keeps the latency close to native, increasing it by just 3.5% at 10K req/s (and even less for lower throughputs).

3.4 Scaling with Load Balancers

In the previous section we showed how *FreeDA* is able to hide the high latency introduced by the dynamic analysis tool ASan. Now, by using load balancing techniques we can easily and out-of-the-box scale throughput. Of course load balancing must be supported by the application under experimentation.

Load balancing Nginx. We deployed *FreeDA* behind a load balancer using Nginx with ASan. Figure 3.4 shows our experimental setup. M1 runs *wrk2* and the HAProxy load balancer (only one pair). *wrk2* has the same configuration as before: 50 seconds run with 4 threads and 80 open connections. M2 and M3 run an instance of Nginx each, serving a 2KiB file containing random data with protocol compression enabled.

Figure 3.7 presents the results, which are similar to those of previous experiments: *FreeDA* is able to hide the overhead of ASan successfully, while also maintaining a higher throughput than ASan. Note that, even though we doubled the resources available in this experiment, we do not observe a 2x increase in throughput as load balancing itself introduces overhead.

Load balancing Redis. We conducted a similar load balancing experiment using Redis. We again used the setup shown in Figure 3.4, with M1 running the Twemproxy load balancer and the *memtier* benchmark, and M2 and M3 running an instance of Redis each. We first tried using a single pair of Twemproxy and *memtier*,

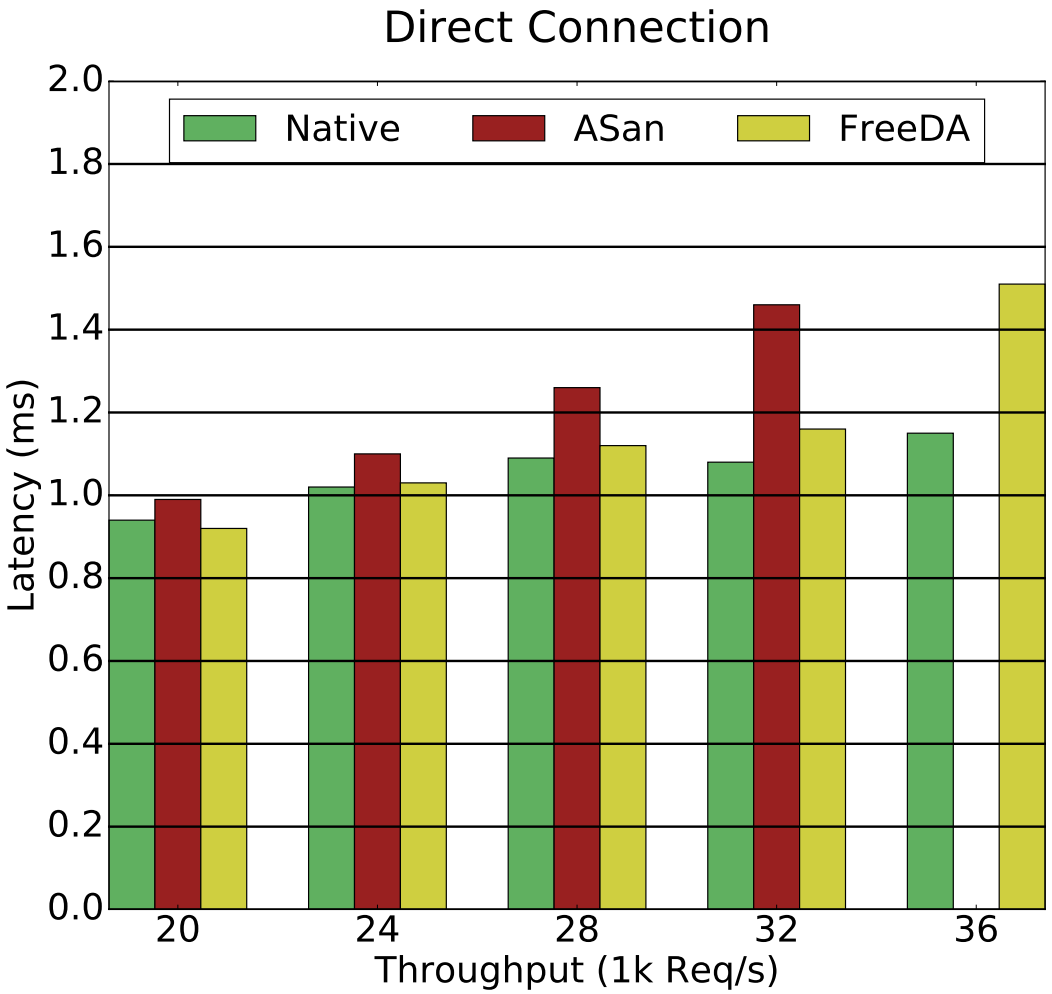

Figure 3.5: Latency of Nginx deployed natively, with ASan, and with ASan through *FreeDA*. Results for a single instance with minimal configuration.

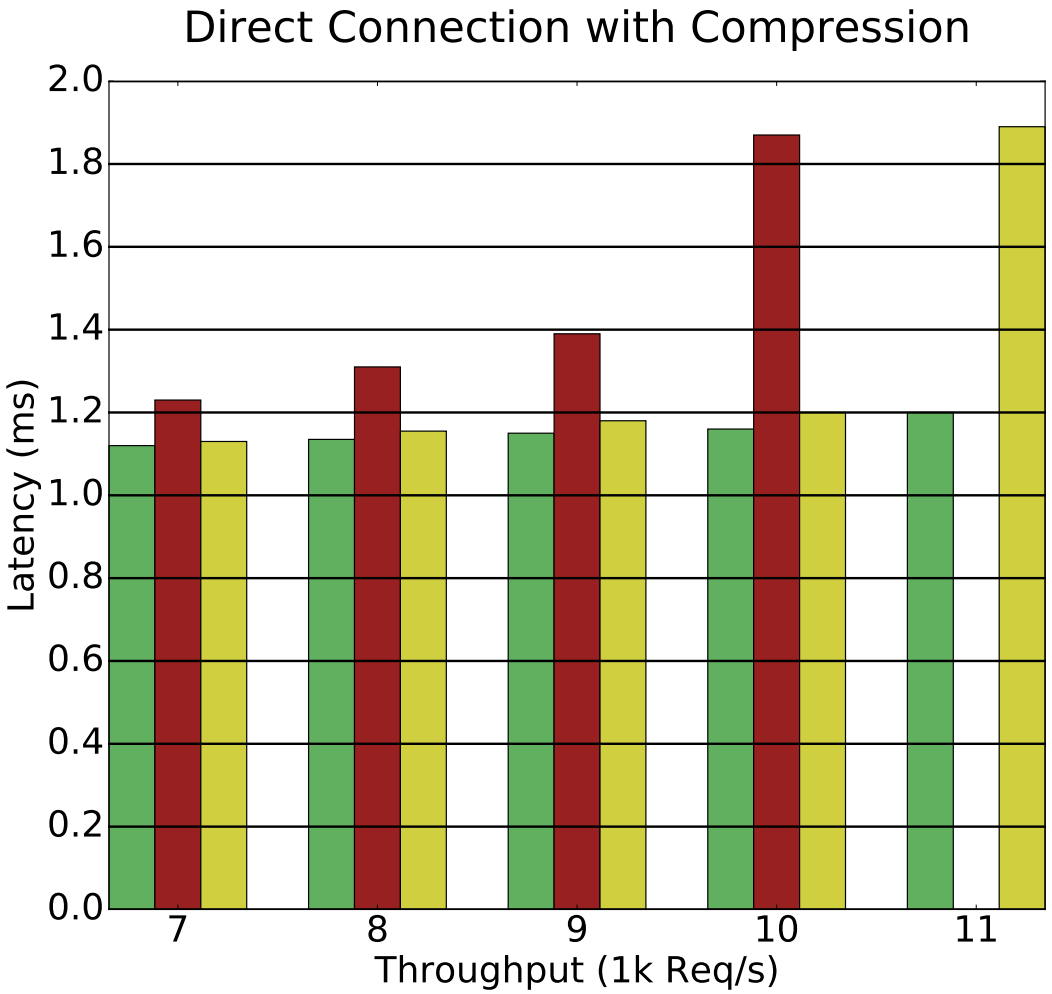

Figure 3.6: Latency of Nginx deployed natively, with ASan, and with ASan through *FreeDA*. Results for a single instance and protocol compression (*Gzip*).

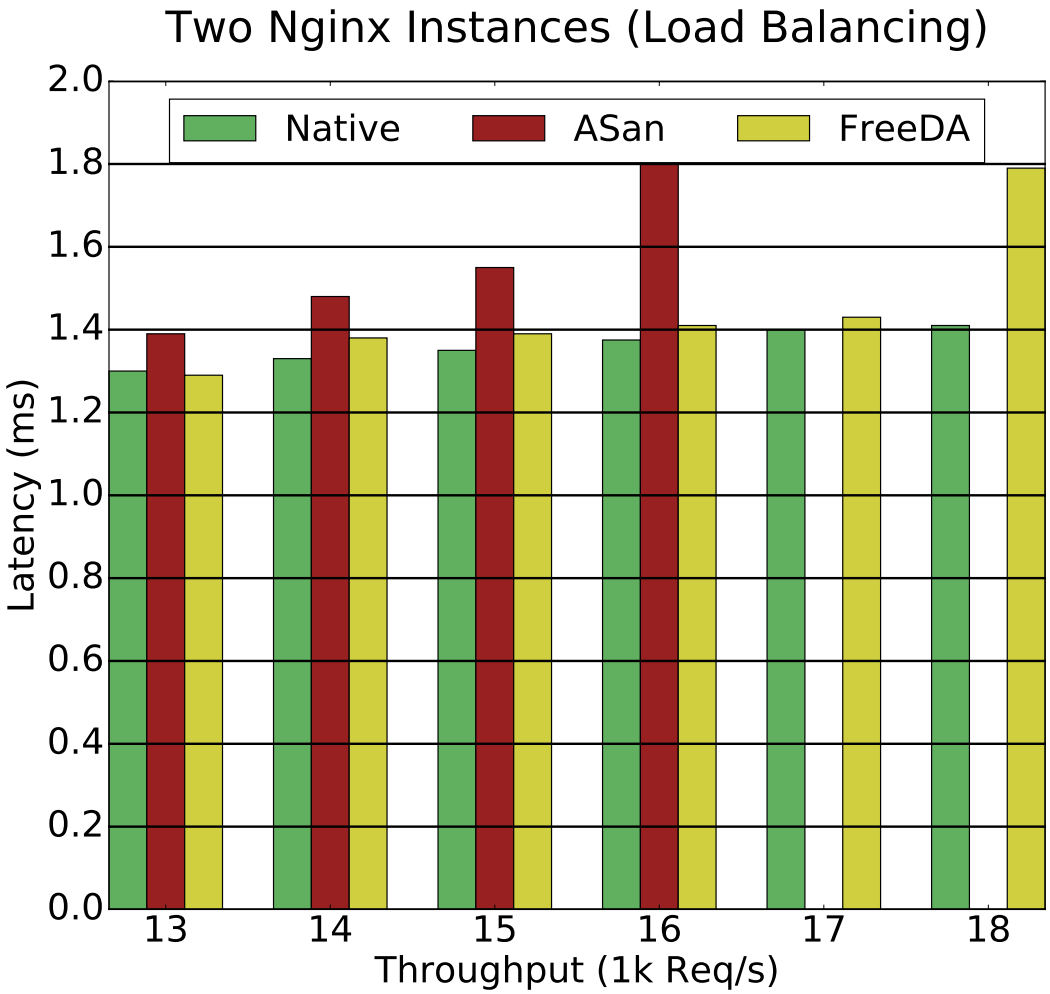

Figure 3.7: Two Nginx instances behind a load balancer with protocol compression (*Gzip*).

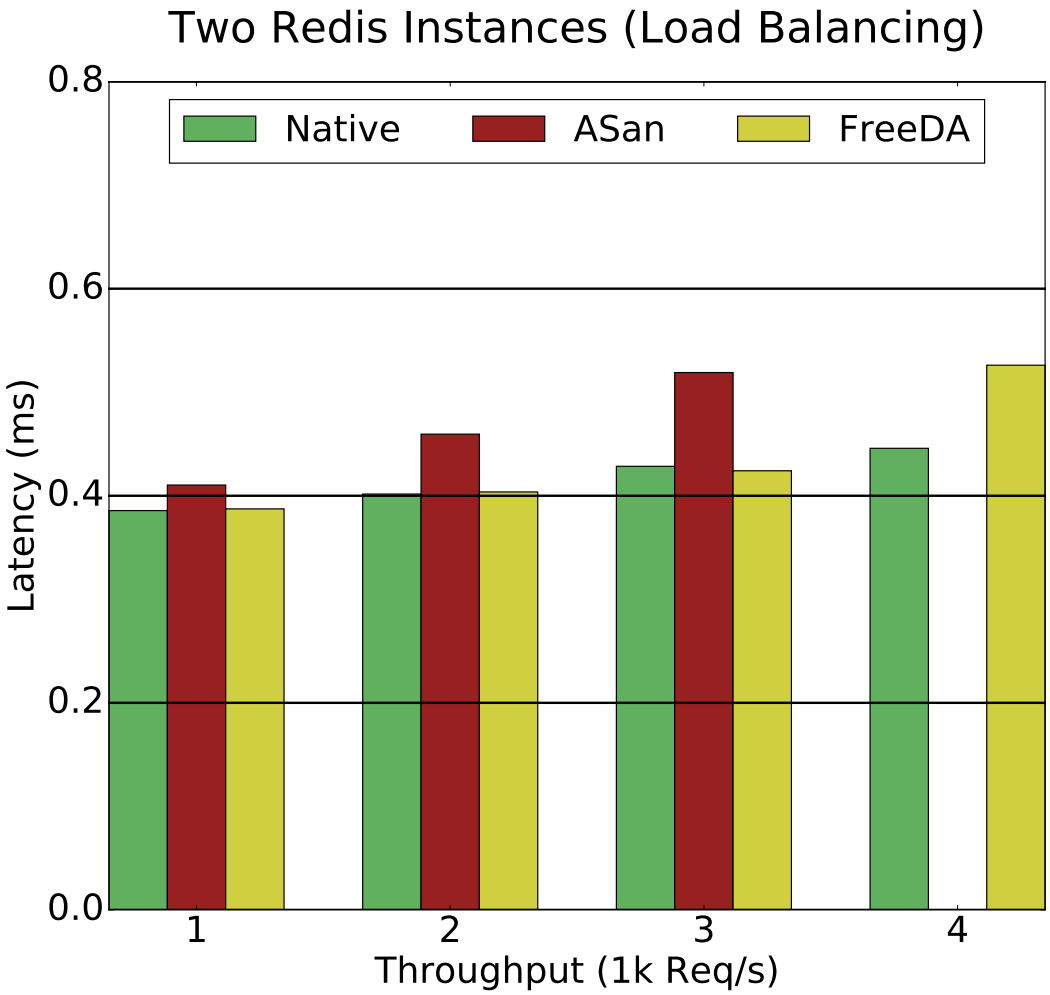

Figure 3.8: Two Redis instances behind a load balancer.

but we were not able to saturate the Redis servers.

Twemproxy was the bottleneck, reaching a maximum of 14K req/s. As a result, we used three pairs of *memtier*/Twemproxy, each achieving a throughput of 10K req/s, for a total of 30K req/s. We configured each *memtier* instance to issue the same number of `GET` and `SET` operations with values of 100 bytes, using 1 thread and 50 connections.

Figure 3.8 shows the results. We observe the same behavior as in the Nginx experiment, with *FreeDA* able to hide ASan's latency, and also achieve a slightly higher throughput.

3.5 Sampling with Weighted Load Balancers

Scaling applications that inherently have small throughput is challenging as it requires many instances and hence a lot of hardware. For example to reach the throughput of a single native Nginx instance, we need to combine approximately 20 instances of Nginx under *Valgrind* behind a load balancer.

To avoid the use of 20 instances of *FreeDA* (which would increase the hardware utilisation cost substantially) we leverage *server weights*, a common feature of load balancers that allows users to split traffic unevenly between backends. In our case, *server weights* configures the load balancer to redirect only a portion of the requests to a *FreeDA* instance and hence to the dynamic analyses tool. In a sense, this trick allows us to *sample* the traffic.

Of course, sampling traffic means that we may miss bugs, thus there is a trade-off between the analyzed traffic and utilization cost that users needs to consider.

In our *weighted load balancing* experiment, we used *FreeDA* with Nginx and Valgrind. After measuring that Valgrind is able to maintain a maximum throughput of $\frac{1}{20}$ of the native's, we configured HAProxy to send $\frac{1}{20}$ of the total traffic to a *FreeDA* instance, and the remaining $\frac{19}{20}$ to a native Nginx instance. We used the same methodology as in the previous experiment, serving a 2KiB file of random data with protocol compression enabled.

Figure 3.9 shows the maximum latency reported by *wrk2*. We did not include the average latency because the number of requests reaching Valgrind is small enough to not change the average latency significantly. The maximum latency of just using Valgrind behind a load balancer shows a severe increase. On the other hand the maximum latency of using Valgrind with *FreeDA* behind a load balancer is close to that of the native version.

3.6 Moving further from Dynamic Analyses

Our architecture is applicable beyond bug-finding dynamic analyses such as Valgrind and compiler sanitizers, and could be used to deploy other types of analyses, e.g. collecting detailed trace logs during production runs.

We conducted two small scale experiments as proofs of concept. In the first experiment, we were able to deploy a Redis instance that keeps data only in-memory as a leader, and a Redis persistent instance that writes its data in a file on the local file system as a follower. Redis persistent was 5x times slower than Redis in-memory due to its delay of writing data to the file. During this experiment the user had no experience of the delay when *FreeDA* was used, and as long as the throughput remains less than the maximum throughput supported by the slowest follower (in our case Redis persistent). We can easily use again the aforementioned load balanced techniques to scale throughput.

It is obvious that the follower's functionality of writing the data to a file it does not exist in the leader instance. To handle the expected differences in system calls (i.e. divergences), we used the feature we discussed in 2.1.2, and also programmatically (in *FreeDA*) tolerated some extra `gettimeofday()` system calls that Redis introduces to write some additional information when initializes the storage file.

In our second experiment we used as a leader an Nginx instance with no logging capabilities (both access and error logs were turned off), and as a follower an Nginx instance that was reporting *debug* level logs. In a similar manner to the above we

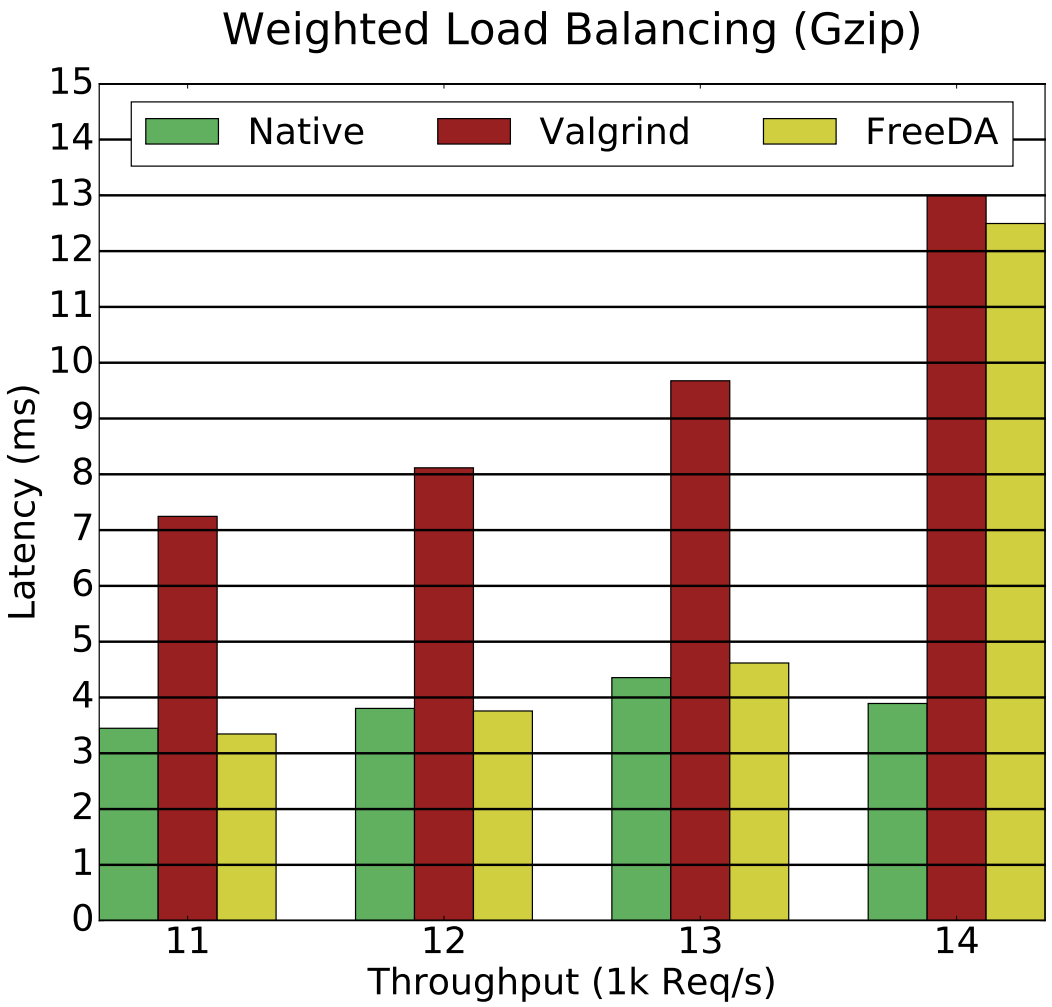

Figure 3.9: Two Nginx instances behind a load balancer that redirects only 1/20 of the requests to a Valgrind instance. The bars show the maximum latency (as expected, the average latency values are similar).

had to handle divergences and the user was experiencing the speed of the fast leader even though the follower was pedantically reporting all internal actions of Nginx.

FreeDA adds some small but measurable overhead to the leader. In our Nginx experiment we observed that if the follower was configured to emit only *error* level logs, then the overhead imposed by *FreeDA* results to a slowdown. In other words, an Nginx instance that reports only *error* level logs is faster than an Nginx instance with no reporting and under *FreeDA*. This is a trade-off that a user must consider as our technique has some performance limitations when it is used with some very lightweight analyses.

We plan to explore similar scenarios in depth in future work.

3.7 Related Work

Both the research community and industry have put a lot of effort into designing many types of dynamic analyses techniques [NS03; Ser+12; SS15; SI09; HJ91; BZ11]. To make an analysis deployable in production, one needs to make it fast. Some types of analysis already meet this criteria. Fast, limited-in-scope analyses such as stack canaries [Cow+98] are now enabled by default in most compilers, while more general analyses such as address sanitization [Ser+12] can already be used in production in some situations, e.g. for certain interactive applications. Prior work has looked at designing dynamic analysis with a low overhead through a variety of means, such as hardware support [Zho+04; OL02] and paralellization [WH07; Nig+08]. DoubleTake [LCB16] proposes a way to restructure common dynamic analysis using *trip wires*, which have a low steady-state overhead. However, it requires existing analyses to be reimplemented in a special way and cannot support analyses that detect out-of-bounds reads bugs like the well-known Heartbleed bug.[8]

The idea of running dynamic analyses in parallel with a native version of the program is an old one. In particular, Patil and Fisher 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 [PF95]. Speck extends this idea by running the native application ahead speculatively and forking multiple analysis instances [Nig+08], and SuperPin by simultaneously executing distinct timeslices of the native program under instrumentation [WH07]. Aftersight employs a record-replay strategy similar to *FreeDA*, but operating at the virtual machine (VM) level [CGC08]. 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 is the first platform that does not implement its own dynamic analyses, and instead supports *existing unmodified* dynamic analysis tools. This separation of concerns between the experts writing the dynamic analyses and the runtime platform that allows their deployment with low overhead is the key con-

[8] `http://heartbleed.com/`

tribution 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 optimisations 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 [Vol+16; XDK12; HC13b; BZ06; Sal+09; MB12; Cox+06; CA78] which extends the record-replay architecture of *Varan* to support low-overhead deployment of out-of-the-box dynamic analysis tools. Unlike prior multi-version execution systems, including *Varan*, in which all versions run at roughly the same speed, *FreeDA* has to support sanitized versions that run significantly slower than the native version.

Variant-based competitive parallel execution uses multi-version execution to increase the performance of a sequential program. The key idea is to create variants of a sequential algorithm with different performance characteristics (e.g. using different heuristics to solve a problem), run them in parallel, and use the results of the algorithm that returns first [TG10]. 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.

Chapter 4

Tolerating Divergences

FreeDA can work out-of-the-box with existing unmodified dynamic analyses tools, and applied to real software but to do so *FreeDA* faces two challenges: (1) the buffer may become full because the sanitized followers are significantly slower than the native leader, and (2) the dynamic analyses deployed in the followers may change the sequence of system calls issued during execution. We already discussed the first challenge in the previous chapter. In this chapter we shall see how we can tolerate system call differences.

As *FreeDA* uses *Varan* under the hood, *Varan* requires by default all program versions to issue the same sequence of system calls. If a follower reaches a point in execution where it issues a system call that does not match the next system call on the shared ring buffer, we say that a *divergence* has occurred and, by default, *Varan* terminates execution of the divergent follower. This is why *FreeDA* extends the properties of *Varan* by being able to reconcile divergences in the sequence of system calls between the leader and each follower through rewrite rules implemented in analysis-specific *reconcilers*.

A *reconciler* is a component in the internal architecture of *FreeDA* which implements a set of rewrite rules [HC15; MB12], specified for each supported dynamic analysis and reused for any program executed with that analysis. Each follower may have its own reconciler which gives us the flexibility to support multiple different dynamic analyses tools simultaneously even if those dynamic analyses tools are incompatible one to another.

Figure 4.1 shows the overview of *FreeDA*'s architecture. Two followers are executed with their own reconciler to tolerate divergences between themselves and the leader.

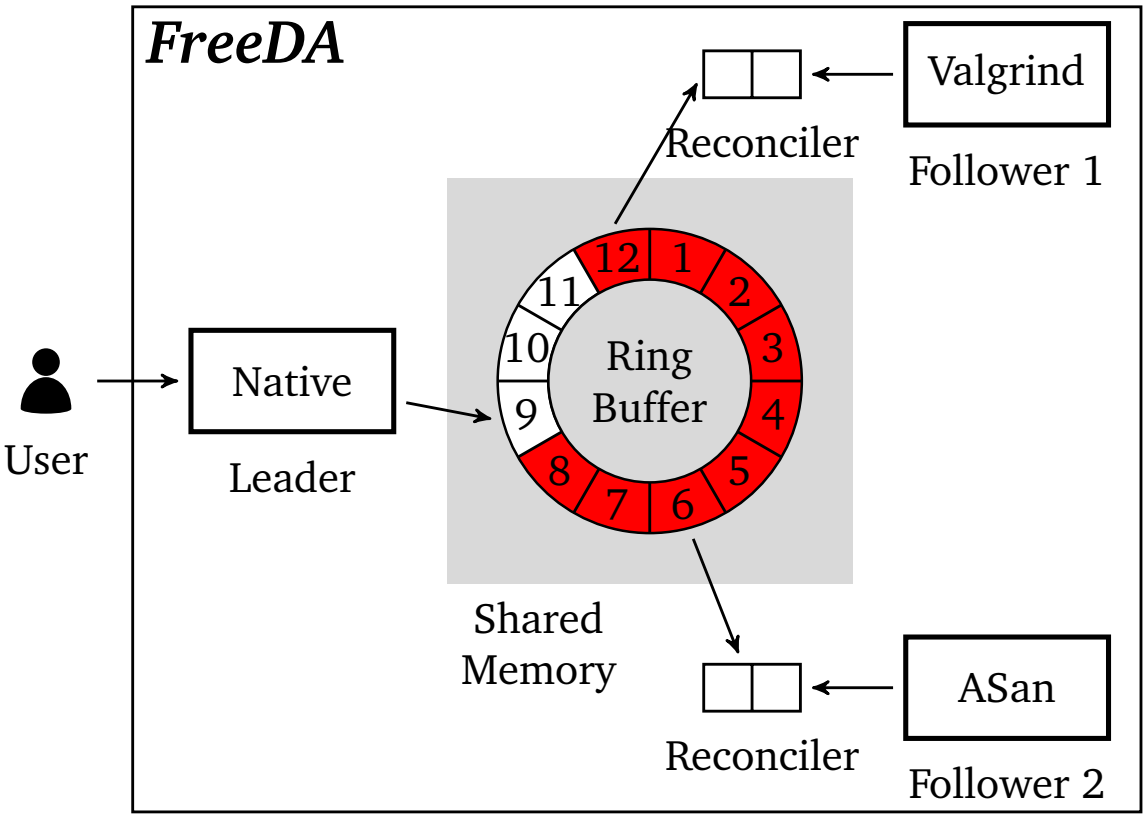

Figure 4.1: Architecture of *FreeDA*. The shared ring buffer grows in clockwise order. Red entries cannot be used by the leader because they contain system calls not yet consumed by all followers. Reconcilers, used from followers to match their system calls with the shared ring buffer.

4.1 Finding Divergences

Currently divergences are found by using `strace` and monitoring the interactions of the programs given some test inputs. After recording the generated system calls in a file for each leader and followers, we use a small custom *Bash* script to separate sequences of system calls from different processes and threads into different files. Then by using `vimdiff` we iterate over pairs of leader-follower processes to extract the expected divergences.

Let us see an example based on how Valgrind augments each system call with additional calls, according to a small set of fixed patterns. Consider an application performing the following sequence of three system calls:

```
read(4, ..., 4096) = 4096
mmap(NULL, 1668976, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f2316379000
rt_sigaction(\sigsegv, {0x555a80, [], SA_RESTORER|SA_STACK, 0x7f932dffb310}, NULL,
    8) = 0
```

Valgrind transforms this sequence as follows (original system calls are highlighted in red):

```
1  getpid()                          = 29387
2  read(1028, "Y", 1)                = 1
3  gettid()                          = 29387
4  write(1029, "Z", 1)               = 1
5  rt_sigprocmask(SIG_SETMASK, [], ~[ILL TRAP BUS FPE KILL SEGV STOP], 8) = 0
6  read(4, ..., 4096) = 4096}
7  rt_sigprocmask(SIG_SETMASK, ~[ILL TRAP BUS FPE KILL SEGV STOP], NULL, 8) = 0
8  mmap(0x405f000, 1668976, PROT_READ, MAP_PRIVATE|MAP_FIXED, 3, 0) = 0x405f000}
9  fstat(3, {st_mode=S_IFREG|0644, st_size=1668976, ...}) = 0
10 readlink("/proc/self/fd/3", "/usr/lib/locale/locale-archive", 4096) = 30
11 stat("/usr/lib/locale/locale-archive", {st_mode=S_IFREG|0644, st_size=1668976, ...})
       = 0
12 mmap(0x802c05000, 16384, PROT_READ|PROT_WRITE|PROT_EXEC,
       MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, 0, 0) = 0x802c05000
```

This example shows how Valgrind transforms most system calls: The original `read` system call is *wrapped* by 6 system calls, between lines 1 and 7. In the first 4 system calls, Valgrind synchronizes itself internally by reading from and writing to file descriptors 1028 and 1029, respectively. Then, Valgrind disables signal delivery on line 5, issues the original system call on line 6, and re-enables signal delivery on line 7. Valgrind wraps in this way most system calls that the program under analysis issues.

System call `mmap`, when used under Valgrind to map a file in memory, provides another example of wrapping: Valgrind issues extra system calls to detect which file was mapped, by gathering information on the file descriptor on line 9, and resolving it to a path on line 10. Then, Valgrind gets information about the file on line 11, and allocates memory to shadow the mapped memory on line 12.

Finally, the system call `rt_sigaction()` installs a signal handler for signal `SIGSEGV`. Valgrind installs a similar handler by its own during initialization, to detect memory errors. When the example application initiates its own `rt_sigaction()`, Valgrind doesn't allow this call to reach the kernel as it will overwrite Valgrind's signal handler. So Valgrind keeps the users signal hanlder internally and the application's system call is never initiated.

For *FreeDA* to be able to allow execution of Valgrind, we need to incorporate

those observations in concrete rules as a reconsiler. Fortunately the only additional effort involved is in writing a reconciler for each analysis. We have already written reconcilers for several popular dynamic analysis tools, and supporting more tools would involve relatively little work.

4.1.1 Future Improvements in Divergence-Finding Tools

As we mentioned before, current we find the expected divergences by using `strace` and `vimdiff`. Unfortunately both of them come with a small set of limitations. Most notably:

- `strace` can not intercept `vDSO` calls like `gettimeofday()`.
- On multithreaded applications, `strace` sometimes splits system calls which makes matching in `vimdiff` harder.
- Separating sequences of system calls for multithreaded applications requires manual intervention.
- Some arguments in system calls are expected to be different (e.g. pointer values) and `vimdiff` reports them as differences.

During our developing time we used workarounds for most of these issues. For instance, *Varan* is able to intercept `vDSO` calls (due to its binary rewriting ability) so to overcome the limitation of `strace`, we slightly modified *Varan* to report all system calls the application does and we recorded them in files.

We envision though an auxiliary tool that could greatly assists developers on finding divergences. Such tool should come with the following properties:

1. Captures every system call made by the application, or ignore specific ones.
2. Ignore specific arguments of system calls.
3. Separate and keep track of multiple processes and threads.
4. Make use of *FreeDA* rewrite rules to report (as differences) only patterns that do not obey the rewrite rules.

The idea is that the user will use this tool to record specific executions of his application under different configurations or dynamic analyses tools and then iteratively will compose rewrite rules until no divergence is found.

Furthermore, we believe it is possible to (partially) automate the process by running programs with and without the analysis and inferring the mapping between the two sequences of system calls. We leave all this as future work.

4.2 Operations

Reconcilers intercept each system call S_f that the follower issues and compares it with the next system call S_b at the shared ring buffer. If they are identical, the reconciler takes the default action of letting *Varan* handle the system calls normally by

dropping S_b from the shared ring buffer and feeding the stored data to S_f. Otherwise if S_f and S_b do not match, we take one of two actions: (A) drop S_b and retry, or (B) execute S_f and leave the shared ring buffer intact. More formally *FreeDA* supports the following operations:

1. **Match.** Returning the stored data from the shared ring buffer to the follower's system call.
2. **Execute.** Allow the follower to execute its system call.
3. **Drop.** Remove the current system call stored at the shared ring buffer.

Interestingly operations *Execute* and *Drop* are enough to tolerate any kind of divergence (note the absence of *Match*). We can informally prove this by continuously dropping S_b until the leader finishes execution, and then execute all S_f. In this way we allow both applications to run independently without any synchronization and thus there can be no divergences. Of course this a pathological case as the core value of NVX is to maximise the use of *Match* during both convergent (both leader and follower issue identical system calls) and divergent execution.

From our experimentation with various applications and dynamic analysis tools, we found that the above operations were sufficient to handle divergences and maximize the use of *Match*. But we can easily invent examples showing that more operations are required. Code 4.1 shows a case where the follower has a reverse order of system calls. With our current set of operations it is impossible to *Match* the system call in line 5 with the one in 15 and simultaneously the one in 6 with the one in 14. We can perform the maximum of one *Match* and that only if we first issue a *Drop*. This kind of reordering implies that we need to extend our set of operations to include new ones that are able to store system calls from the shared ring buffer into axillary memory for future use in the follower's sequence.

Another interesting case is presented by simply asking if every *Match* operation must be followed by a *Drop*. Code 4.2 shows a case where the follower issues a leader's system call one extra time. The only way to match follower and leader, is by not issuing a *Drop* after the first *Match*.

This example raises the question of how can we distinct a case where the follower's second `getpid()` must be matched with one from the leader, and a case where we want to execute it instead. Currently the only way to know this is from the user. This might be a very challenging scenario for creating automatic tools that extract divergence patterns as we discussed in Section 4.1.1. Up to now, we have not observed any such case.

Listing 4.1: A follower with a reverse order of system calls

```
1 // Leader's code
2 int main(int argc, char **argv) {
3   struct timeval tv;
4
5   getpid();
6   gettimeofday(&tv, NULL);
```

```
  return 0;
}

// Follower's code
int main(int argc, char **argv) {
  struct timeval tv;

  gettimeofday(&tv, NULL);
  getpid();
  return 0;
}
```

Listing 4.2: A follower with consecutively similar system calls

```
// Leader's code
int main(int argc, char **argv) {
  getpid();
  return 0;
}

// Follower's code
int main(int argc, char **argv) {
  getpid();
  getpid();
  return 0;
}
```

4.3 Types of Divergences

Now that we have a set of operation we can systematically create combinations of them to synthesize divergence types. First we categorize divergences based on their demand of auxiliary memory. If the follower's divergence can be handled only by using *Execute* or *Drop* operations, then we say that the divergence is *Plain*, else if we need to use extra memory (i.e. use additional operations) then it is *Complex*. Note that we do not include *Match* as it is not considered as a divergence handling operation.

We propose the following types for expected divergences:

1. **Plain.** Divergences of this type can be handled by the sole use of *Execute* and *Drop*.

   (a) *Additive.* When the follower issues additional system calls non-existing to the leader. Can be handled with the use of *Execute*.

(b) *Subtractive.*[1] When the follower skips leader's system calls. Can be handled with the use of *Drop*.

(c) *Continuative.* When the follower repeats a leader's system call multiple consecutive times. Can be handled with the absence of a *Drop* after a *Match*.

2. **Complex.** Divergences of this type require auxiliary memory to store system calls from the shared ring buffer for revisiting them later.

(a) *Reordering.* When the follower has rearranged the order of leader's system calls. See Code 4.1.

(b) *Mixing.* When the follower issues additional system calls the need to be matched and they are not consecutive. See Code 4.3.

We leave further analysis for *Complex* divergences as future work.

Listing 4.3: A follower with a *Mixing* divergence

```
// Leader's code
int main(int argc, char **argv) {
  struct timeval tv;

  getpid();
  gettimeofday(&tv, NULL);
  return 0;
}

// Follower's code
int main(int argc, char **argv) {
  struct timeval tv;

  getpid();
  gettimeofday(&tv, NULL);
  getpid();
  return 0;
}
```

From our experiments presented in Chapter 3, we found various patterns that can be explained more systematically with our proposed divergence types:

1. Dynamic analyses usually introduce extra system calls in the beginning and end of the analyzed program, to initialize the internal state of the analysis itself, perform sanity checks, or gather statistics after the program ends.

[1] Fun Fact: Subtractive can be Additive (and reverse) if we swap roles of leader and follower.

2. System calls of the program under analysis are sometimes preceded and/or followed by system calls of the analysis tool. For instance, Valgrind surrounds most system calls with two extra system calls: one to disable signal delivery just before the original system call, and another to re-enable them immediately after. Or, ASan adds a `sched_getaffinity` after a `fork`.
3. Dynamic analysis tools have additional functionality which does not exist in the target program. The system calls invoked by this additional functionality may not have a specific pattern as described in pattern 2. For instance, if a non-fatal memory error occurs, ASan reports it by writing information to a text file. The system calls that open, write, and close such a file do not occur in the original program.
4. Analyses typically emulate some system calls without issuing them. For instance, most analyses monitor some signals (e.g. `SIGSEGV`) by installing signal handlers. As a results, if the program under analysis registers its own handler, the dynamic analysis just updates its internal state without issuing any signal-handling related system call.
5. In the experiment 3.6, Redis configured with persistent storage was issuing two consecutive `gettimeofday()` in which one was not present in the leader.

Pattern 1 is mainly *Additive* but it can also be *Subtractive* in cases where a dynamic analyses tool does not require a specific library that the original program does. Pattern 2 can be broken down to two *Additive* divergences with a *Match* in between. Pattern 3 is the purest form of *Additive* divergence. Pattern 4 is *Subtractive*. And pattern 5 is *Continuative*. We did not observe any *Complex* divergence, though we speculate that experiments which include applications of different versions most probably will expose *Complex* divergences.

4.4 A DSL for Expected Divergences

In order to allow users to easily express expected divergences in *FreeDA*, we created the Domain-Specific Language (DSL): *Divela*.[2,3] The key idea behind *Divela* is that expected divergences can be expressed as a mapping between system calls of the leader and system calls of the follower. Code 4.4 shows a simple example that matches a single `getpid()` issued by the leader, to two from the follower. *Divela* accepts the code in 4.4 as input and generates a *C* library that *FreeDA* loads as a *reconciler*.

The generated library exposes an API that accepts the currently executing system call of the follower and the one available at the shared ring buffer, and it returns the appropriate *operation* that *FreeDA* has to issue to tolerate the divergence. For instance, in the example 4.4 the reconciler will return one *Drop* and two *Execute*. To be noted that there will be no *Match*.

[2] `https://github.com/daniel-grumberg/varan-div`
[3] The name comes from **dive**rgence and **la**nguage.

Listing 4.4: Simple example of *Divela*

```
1 getpid() => getpid(),
2             getpid()
```

For *Divela* to return a *Match* the `self()` keyword is required. `self()` maps a system call from the ring buffer with one from the follower. In Code 4.5 we can see various examples of using `self()`. Case 1 is as we described above; the reconciler issues a *Drop* and two *Execute* operations. Case 2 will issue a *Match*, a *Drop* and an *Execute*. Case 3 is similar to case 2 but with a different order of operations. Case 4 should issue two *Match* operations and a single *Drop*. Case 5 uses another keyword of the language: `nothing`, which just issues a *Drop*. `nothing`'s purpose is to handle *Subtractive* divergences.

Internally, *Divela* generated reconcilers use state automata to keep track of the system calls issued and which should be next. If follower's next system call is not expected by the internal automaton of the reconciler, then we report an error of an unexpected divergence, else the automaton returns the appropriate operation to be issued by *FreeDA*.

Until now we discussed system call divergences at a very high level. But in real systems checking if the follower's and leader's system calls are of the same kind is not enough. Parameters are also important. This is why *Divela* supports *Hooks* which are C code blocks operating on the system call parameters. Code 4.6 shows an example of a read system call. If *FreeDA* finds a `getpid()` at the shared ring buffer and the follower issues a `read()`, then this event is a divergence if $a > 200$.

4.4.1 *Divela* Extensions

Divela is in its infancy and still under heavy development. Many features and mechanisms are still missing with most notably the fact that we only partially support the *Additive* divergence type. In this section we propose some new features that extend the expressiveness of *Divela*.

1. `nothing` **on the left-hand side.** By adding `nothing` on the left-hand side we have full support of *Additive* divergences.

2. **Supporting *Ignoring-File-Paths*.** In Section 2.1.2 we discussed how *Varan* parses `open()` system calls and keeps track of file descriptors throughout the execution. To support this mechanism in *Divela* we require two additional features: (a) a way to keep global variables that can be accessed through *Hooks* and thus different *Divela* rules can be cross synchronized, and (b) access the return value of a system call. Feature 1 is also mandatory as the user should be able to map all extra system calls made by the follower to `nothing`. Ideally we could even provide special keywords to auto-generate system calls that operate on file descriptors.

Listing 4.5: Examples that use *Match*

```
// Case 1
getpid() => getpid(),
            getpid()

// Case 2
getpid() => self(),
            getpid()

// Case 3
getpid() => getpid(),
            self()

// Case 4
getpid() => self(),
            self()

// Case 5
getpid() => nothing
```

Listing 4.6: Examples C code blocks in *Divela*

```
getpid() => read(a, _, _) { if ($(a) < 200) return 0; },
            getpid()
```

Listing 4.7: Multiple rules example

```
1 getpid() => getpid(),
2             getpid()
3
4 getpid() => getpid(),
5             open(),
6             getpid()
```

3. **Multiple system calls on the left-hand side.** At the moment, *Divela* does not support *Complex* divergences. Adding multiple system calls on the left-hand side would be the first step towards *Complex* divergences.

4. **Multiple automata invocation.** Or multiple rule invocation. When divergences have the same left-hand side and *prefix* of system calls of the right-hand side, *Divela* reconcilers need to keep track multiple sequences concurrently until the point where they differ. Currently example Code 4.7 is not supported in *Divela*.

Chapter 5

Conclusion

This report presented *FreeDA*, a system that enables deploying expensive dynamic analyses while providing close to native performance. *FreeDA* leverages *Varan*, a multi-version execution system, and *Divela*, a DSL that expresses divergences in a convenient manner, to run the native application along with several existing, unmodified dynamic analyses versions.

Besides masking the overhead of slow dynamic analyses, *FreeDA* can also deploy analyses that are incompatible with one another, such as Valgrind analyzing the same execution as the address and memory compiler sanitizers.

We tested *FreeDA* on realistic scenarios like high-performance server applications. We found that *FreeDA* can run existing analyses while masking the overhead that would otherwise make these analyses prohibitive.

FreeDA is the first system that can deploy existing unmodified analyses in production with low overhead, and as such we believe it is an important step in making current software more reliable.

Bibliography

[CA78] Liming Chen and Algirdas Avizienis. “N-version programming: A fault-tolerance approach to reliability of software operation”. In: *Digest of Papers FTCS-8: Eighth Annual International Conference on Fault Tolerant Computing*. 1978, pp. 3–9.

[Neu+80] Peter G Neumann et al. “A provably secure operating system: The system, its applications, and proofs”. In: *Computer Science Laboratory Report CSL-116, Second Edition, SRI International* (1980).

[KL86] John C Knight and Nancy G Leveson. “An Experimental Evaluation of the Assumption of Independence in Multiversion Programming”. In: *IEEE Transactions on software engineering* 12.1 (Jan. 1986), pp. 96–109. ISSN: 0098-5589. DOI: 10.1109/TSE.1986.6312924. URL: http://dx.doi.org/10.1109/TSE.1986.6312924.

[BKL90] Susan S. Brilliant, John C. Knight, and Nancy G. Leveson. “Analysis of faults in an N-version software experiment”. In: *IEEE Transactions on software engineering* 16.2 (1990), pp. 238–247.

[KL90] John C Knight and Nancy G Leveson. “A reply to the criticisms of the Knight & Leveson experiment”. In: *ACM SIGSOFT Software Engineering Notes* 15.1 (1990), pp. 24–35.

[HJ91] Reed Hastings and Bob Joyce. “Purify: Fast detection of memory leaks and access errors”. In: *In proc. of the winter 1992 usenix conference*. Citeseer. 1991.

[LT93] Nancy G Leveson and Clark S Turner. “An investigation of the Therac-25 accidents”. In: *Computer* 26.7 (1993), pp. 18–41.

[PF95] Harish Patil and Charles N Fischer. “Efficient Run-time Monitoring Using Shadow Processing.” In: *AADEBUG*. Vol. 95. 1995, pp. 1–14.

[Cow+98] Crispan Cowan et al. “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks.” In: *Usenix Security*. Vol. 98. 1998, pp. 63–78.

[OL02] Jeffrey Oplinger and Monica S Lam. *Enhancing software reliability with speculative threads*. Vol. 37. 10. ACM, 2002.

[Tas02] Gregory Tassey. “The economic impacts of inadequate infrastructure for software testing”. In: *National Institute of Standards and Technology, RTI Project* 7007.011 (2002).

[NS03] Nicholas Nethercote and Julian Seward. “Valgrind: A Program Supervision Framework”. In: *Electronic Notes in Theoretical Computer Science* 89.2 (2003).

[Zho+04] Pin Zhou et al. “iWatcher: Efficient architectural support for software debugging”. In: *ACM SIGARCH Computer Architecture News*. Vol. 32. 2. IEEE Computer Society. 2004, p. 224.

[BZ06] Emery D Berger and Benjamin G Zorn. “DieHard: probabilistic memory safety for unsafe languages”. In: *Acm sigplan notices*. Vol. 41. 6. ACM. 2006, pp. 158–168.

[Cox+06] Benjamin Cox et al. “N-Variant Systems: A Secretless Framework for Security through Diversity.” In: *Usenix Security*. Vol. 6. 2006, pp. 105–120.

[NFA06] Lajos Nagy, Richard Ford, and William Allen. “N-version programming for the detection of zero-day exploits”. In: (2006).

[BH07] Luiz André Barroso and Urs Hölzle. “The case for energy-proportional computing”. In: (2007).

[WH07] Steven Wallace and Kim Hazelwood. “Superpin: Parallelizing dynamic instrumentation for real-time performance”. In: *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2007, pp. 209–220.

[CGC08] Jim Chow, Tal Garfinkel, and Peter M Chen. “Decoupling dynamic program analysis from execution in virtual environments”. In: *USENIX 2008 Annual Technical Conference on Annual Technical Conference*. 2008, pp. 1–14.

[Nig+08] Edmund B Nightingale et al. “Parallelizing security checks on commodity hardware”. In: *ACM Sigplan Notices*. Vol. 43. 3. ACM. 2008, pp. 308–318.

[Sal+09] Babak Salamat et al. “Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space”. In: *Proceedings of the 4th ACM European conference on Computer systems*. ACM. 2009, pp. 33–46.

[SI09] Konstantin Serebryany and Timur Iskhodzhanov. “ThreadSanitizer: data race detection in practice”. In: *Proceedings of the Workshop on Binary Instrumentation and Applications*. ACM. 2009, pp. 62–71.

[TG10] Oliver Trachsel and Thomas R Gross. “Variant-based competitive parallel execution of sequential programs”. In: *Proceedings of the 7th ACM international conference on Computing frontiers*. ACM. 2010, pp. 197–206.

[BZ11] Derek Bruening and Qin Zhao. “Practical memory checking with Dr. Memory”. In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2011, pp. 213–223.

[CH12] Cristian Cadar and Petr Hosek. “Multi-version software updates”. In: *Proceedings of the 4th International Workshop on Hot Topics in Software Upgrades*. IEEE Press. 2012, pp. 36–40.

[MB12] Matthew Maurer and David Brumley. “TACHYON: Tandem execution for efficient live patch testing”. In: *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. 2012, pp. 617–630.

[Ser+12] Konstantin Serebryany et al. “AddressSanitizer: a fast address sanity checker”. In: *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 2012, pp. 309–318.

[XDK12] Hui Xue, Nathan Dautenhahn, and Samuel T King. “Using replicated execution for a more secure and reliable web browser.” In: *NDSS*. 2012.

[Cav13] Mark Cavage. “There’s just no getting around it: you’re building a distributed system”. In: *Queue* 11.4 (2013), p. 30.

[HC13a] Petr Hosek and Cristian Cadar. “Safe software updates via multi-version execution”. In: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pp. 612–621.

[HC13b] Petr Hosek and Cristian Cadar. “Safe software updates via multi-version execution”. In: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pp. 612–621.

[HC15] Petr Hosek and Cristian Cadar. “Varan the Unbelievable: An efficient N-version execution framework”. In: *ACM SIGARCH Computer Architecture News* 43.1 (2015), pp. 339–353.

[Lag+15] Ignacio Laguna et al. “Debugging high-performance computing applications at massive scales”. In: *Communications of the ACM* 58.9 (2015), pp. 72–81.

[Qia+15] Weizhong Qiang et al. “MUC: Updating cloud applications dynamically via multi-version execution”. In: *Future Generation Computer Systems* (2015).

[SS15] Evgeniy Stepanov and Konstantin Serebryany. “MemorySanitizer: fast detector of uninitialized memory use in C++”. In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2015, pp. 46–55.

[LCB16] Tongping Liu, Charlie Curtsinger, and Emery D Berger. “DoubleTake: fast and precise error detection via evidence-based dynamic analysis”. In: *Proceedings of the 38th International Conference on Software Engineering*. ACM. 2016, pp. 911–922.

[Vol+16] Stijn Volckaert et al. “Secure and Efficient Application Monitoring and Replication”. In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 2016.